THE UNIVERSITY OF ALBERTA

# GENERAL PRINCIPLES OF SOFTWARE DESIGN IN A GRAPHICAL SUBROUTINE PACKAGE

by

William C. Jackson



A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

Spring, 1972

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled GENERAL PRINCIPLES OF SOFTWARE DESIGN IN A GRAPHICAL SUBROUTINE PACKAGE submitted by William C. Jackson in partial fulfillment of the requirements for the degree of Master of Science.

## ABSTRACT

As computer software becomes increasingly more complex, the need for an adequate and widely accepted set of software design principles becomes critical. Traditionally, the emphasis in computer science has been on the algorithm rather than on good software design. General principles for the design of computer software are very important but particularly difficult to define, and discussion of general principles is almost entirely absent from the literature. In this thesis, general principles for software design are discussed, and examples of their influence on the redesign of a graphical subroutine package are described.

# ACKNOWLEGEMENTS

I would like to express my appreciation to my supervisor, Dr. J.P. Penny, for his guidance and encouragement in the preparation of this thesis; to K.F. May for his helpful suggestions and continued interest in the project; and to G. F. Gabel for the financial assistance which helped make it possible to implement the Graphical Subroutine Package.

TABLE OF CONTENTS


Page

CHAPTER I: INTRODUCTION ..... 1

CHAPTER II: GENERAL PRINCIPLES OF SOFTWARE DESIGN

2.0 Introduction ..... 4

2.1 Modularity ..... 5

2.2 Standardization ..... 11

2.3 Efficiency And Stability ..... 14

2.4 Flexibility and Extensibility ..... 17

2.5 Documentation

2.5.1 Software Specification ..... 19

2.5.2 Internal Logic Documentation ..... 25

2.5.3 User's Guide ..... 27

CHAPTER III: SOFTWARE DESIGN PRINCIPLES ILLUSTRATED IN A GRAPHICAL SUBROUTINE PACKAGE

3.0 Introduction ..... 31

3.1 Modular Construction Of GRIDSUB ..... 32

3.2 Standardization In GRIDSUB ..... 36

3.3 Efficiency vs Stability, GRID-SYSTEM/360 Communication ..... 42

3.4 Flexibility And Extensibility In GRIDSUB ..... 45

3.5 Documentation For GRIDSUB

3.5.1 Software Specification ..... 51

3.5.2 Internal Logic Documentation ..... 52

3.5.3 User's Guide ..... 53

CHAPTER IV: CONCLUSION ..... 58

Page

REFERENCES ..... 62

APPENDIX I: FORTRAN SUBROUTINES - THE GRIDSUB PACKAGE .. 65

APPENDIX II: A SAMPLE PROGRAM USING GRIDSUB ..... 95

APPENDIX III: OS S TYPE CALLING SEQUENCE ..... 100

APPENDIX IV: A SAMPLE OF INTERNAI LOGIC
DOCUMENTATICN ..... 103

APPENDIX V: FILE AND TABLE SPACE USAGE UNDER
GRIDSUB ..... 113

# LIST OF FIGURES


| | | Page |
|---|---|---|
| Figure 2-1 | Tree-Like Structure Cf GRIDSUB Modules .. | 10 |
| Figure 2-2 | Internal Logic Manual Index ..... | 23 |
| Figure 2-3 | IBM Program Logic Manual Organization ... | 24 |
| Figure 2-4 | IBM System Reference Library Organization ..... | 29 |
| Figure 3-1 | Save Area Linkage ..... | 41 |
| Figure 3-2 | A Lower Level In Communication ..... | 44 |
| Figure 3-3 | Block File ..... | 50 |
| Figure 3-4 | GRIDSUB User's Guide Table Of Contents .. | 55 |
| Figure III-1 | Standard Parameter List Convention .... | 102 |
| Figure IV-1 | Packed Format ..... | 105 |
| Figure IV-2 | Flowchart For PACTOBIN/BINTOPAC ..... | 106 |
| Figure IV-3 | Binary To Packed Format ..... | 107 |
| Figure IV-4 | Packed Format To Binary ..... | 108 |
| Figure V-1 | GRIDSUB ..... | 116 |

# CHAPTER I

# INTRODUCTION

The project dealt with in this thesis included the redesign and rewriting of a moderately complex software system. Although this redevelopment consumed a major proportion of the total time spent on the project, the description of the software itself has been relegated to examples in Chapter III and the Appendices. The author's chief interest lies in attempting to formulate general principles for software design. Although software design is in many respects similar to hardware design, for example the need for extensibility and flexibility, this thesis deals exclusively with software design. In Chapter II of this thesis, several aspects of software design are explored with illustrations being drawn from the software system described in Chapter III and the Appendices.

Studies on principles of software design are almost entirely absent from the literature. There are many books which, from their titles, would appear to deal with software design principles and techniques. Upon closer examination however, one finds articles dealing mainly with a new piece of hardware or software. An example is Software Engineering (Tou 1970), a two-volume conference proceedings in which most articles describe new hardware and software systems.

It is doubtful whether software design can be

considered a science; however, software design is very similar to many areas in engineering, that is, both areas attempt to advance the current level of technology. Engineers have developed principles for highway and building construction, and it should be possible to have software engineers formulate software engineering principles.

A major objective for many computer scientists is to develop effective algorithms to solve their problems. Although the problems being investigated vary greatly, the computer scientists have in common the need for well-written software packages for problem solving. A great deal of research has been devoted to algorithm development in such areas as numerical analysis and mathematical programming. However, studies on software design principles are rarely found in the literature.

The lack of adequate software results in much duplication of software packages. For example, a compiler is written, but due to a lack of portability, documentation, or efficiency, it is not utilized at other installations. Instead, another compiler is written, probably no more competently than the first compiler. Neumann (1969) suggests eleven reasons why we are still producing inadequate software. Among these reasons he mentions the tendency 'to repeat the mistakes of others'. This duplication of development and resources could be greatly reduced if more emphasis was placed on software design principles.

The aspects of software design discussed in Chapter II are:

A) Modularity

B) Standardization

C) Efficiency and Stability

D) Flexibility and Extensibility

E) Documentation

Following the general discussion in Chapter II, each of the points is re-examined in Chapter III, with emphasis placed on the effect these principles have had on the redesign of a graphical subroutine package, GRIDSUB (Huen 1969).

# CHAPTER II

# GENERAL PRINCIPLES CF SOFTWARE DESIGN

## 2.0 Introduction

This chapter formulates general principles of software design. The few articles that exist on software design principles do not cover these principles in general, but concentrate cn specific aspects. Fcr example, 'How To Write Software Specifications' (Hartman and Owens 1967) covers only one aspect of software design, as the title implies.

Hartman and Owens aptly describe the situation in their opening paragraph: 'In general, computer software is getting more complicated at an increasing rate. Unfortunately, the people who develop software have been at it for only a relatively shcrt time. The result is predictable: ever-larger software troubles. This, in turn, leads many people to feel insecure about software development, to think of it as a modern "black art" for which even the most able practitioners lose the recipe every few months.'

Good practices in design of software are very important, but hard to define. This chapter is not intended to be a recipe for software design, but rather a discussion of some important principles and practices: modularity, standardization, efficiency, stability, flexibility, extensibility, and documentation.

## 2.1 Modularity

Sharp (1970) distinguishes between software engineering and software architecture, emphasizing that 'little or no regard is being paid to architecture'. It is the architect who designs software. He writes the software specification, and decides how the system is to be constructed by the programmers. The major factor determining how a system will be constructed is the modular structure of the system outlined by the software architect. It is the degree to which the architect should make the software modular that is of concern here.

In the simplest case, a module may be defined as a closed subroutine. Sperry Rand (UNIVAC 1959) defines a closed subroutine as: 'a subroutine not stored in the main path of the routine. Such a subroutine is entered by a jump operation and provision is made to return control to the main routine at the end of the operation. The instructions related to the entry and re-entry constitute a linkage.'

The idea of modularity at many levels is expressed in the IBM (1970-4) definition of a program module: 'The input to, or output from, a single execution of an assembler, compiler, or linkage editor; hence, a program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading.'

Software should be modular at a number of levels. Modularity allows the programmer to be concerned with one

part of the software at a time. This makes software development easier and facilitates changes to the system by modular replacement. A module may be viewed as a single, very powerful operation. The highest level module is the complete system itself. The designer can therefore create a structure of modules, with many small logical units combining to form the total system. Figure 2-1 is an example of this type of structure within GRIDSUB. An explanation of the modules mentioned in this figure will be found in Appendix I.

In addition to the modules shown in Figure 2-1, there are modules which are referenced by other modules at the same or higher levels. One such module, XANDY, used to calculate co-ordinates for GRID code, is referenced by most of the Block Definition Routines.

Modularity must be apprcached carefully and with considerable forethought. For each module, the designer needs to describe carefully:

(1) Interface: The interface consists of linkage between modules and communication between modules. Linkage involves passing and returning control, whereas communication involves the passing of parameters. The interface allows the user to treat the module or group of modules being called as a single, very powerful instruction.

(2) Functional Specification: The functional

specification is a detailed description of the information (parameters) required by the module; the operations the module is to perform; the parameters set or returned by the module, and the modules, if any, referenced by the module being described.

An example of modularity increasing system efficiency in a restrictive environment is given below. Consider the case where the software performs many repetitive operations. Straight-line coding might not be acceptable because of limitations of core size. However, the operation could be programmed as a module, thus reducing the overall size of the software. The software would then consist of a relatively small mainline program that calls the many modules; probably some of these modules would be used only rarely, while others would be called a great deal. The modular form of a program can reduce the system overhead in a paging environment by reducing page faulting. Page faulting occurs when a reference is made to a page which is not in core. When a page fault cccurs, the operating system removes a page and brings in the required page. Obviously, excessive page faulting can greatly increase software overhead. By organizing the mcdules in such a manner that the mainline and the most used mcdules are loaded together, they will tend to be loaded on the same page or group of pages. Most paging algorithms will not remove from core pages that have a high number of references to them unless absolutely necessary. Hence, page faulting is likely to

occur only on the pages containing the least frequently used modules, and thus the paging overhead should be minimal.

Modularity is also of value in non-paging environments, especially when overlaying is used. Overlaying is a technique for bringing routines into core from external storage as they are needed, so that several routines will occupy the storage but at different times. This is used when memory requirements exceed available memory. It is generally up to the programmer to determine the overlay structure. The overhead involved with overlaying can be kept to a minimum by carefully setting up the overlay structure so that, if possible, modules used a great deal are not overlaid, or if they are overlaid, it is only with routines that are used infrequently. In this manner the number of overlay operations required should be greatly reduced.

Another advantage of a modular system is the ability to construct a program library. The modules of the system are combined to form a single data set which includes a directory for identifying and locating the modules in the library. The loader or linkage editor program supply the modules needed by the user's program from the library. The loader includes only the modules referenced. This can result in a considerable saving in space for the user who is using only a subset of the entire system contained in the library.

Modularity must also be approached with caution. It is very easy to make a system too modular. If every module is a closed subroutine then a register save and restore will be performed every time the subroutine is used. If the module is rarely used, it may not be practicable to implement the routine as a separate subroutine with its own entry and exit, but it would be preferable tc add the necessary code to the routines requiring this function.

Another important consideration is the compilation or assembly charge for small modules. Since most assemblers and compilers have a substantial initialization cost compared to the cost of compiling a small subroutine, it may be considerably cheaper to combine several subroutines in one assembly or compilation.

In summary, programming a large software system as a set of clearly defined modules is essential because the project can be broken down into more manageable pieces, thereby allowing group development on small pieces of software that are clearly defined. Changes in functional requirements or hardware specifications, either during development or later, may involve changes only to isolated modules. Finally, modular design of software may increase efficiency and simplify the software in a restrictive environment, where the total system is too large for the environment at any one time and the use of overlay structures or paging is required.

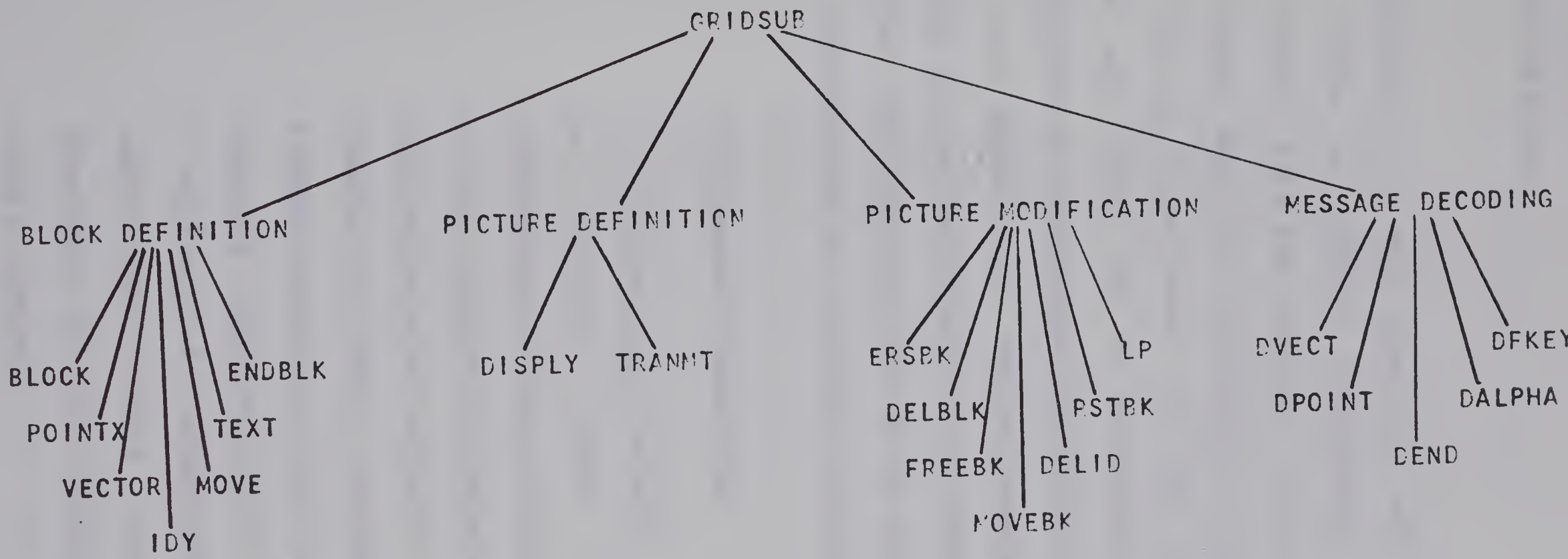



Figure 2-1 Tree-Like Structure of GRIDSUB Modules

## 2.2 Standardization

Maintenance is a very important and large job which normally falls on systems programmers who have had no previous contact with the software. This unfamiliarity with software arises when one installaticn or manufacturer writes software that is distributed to many installations. In order to make this job as straightforward as possible, standardization of programming practices is important. For many of the areas that require standardization, certain conventions will be laid down in the software specification. This section will describe some cf the things that should be standard practice for all software designers.

### (1) Linkage Conventions

Linkages are normally standardized for an operating system. By following the standardized linkage convention for system, it is relatively simple to rewrite or add a subroutine to a software system, with few changes to other routines. A standardized linkage convention facilitates updating and changing modules within a large system, with a minimum of interference among other subroutines. This also allows modules developed for one system to be used in another system.

(2) Programming Practices

Whenever possible symbolic addressing should be used, even for absolute expressions. This can be very helpful for debugging and when making alterations if the assembler or compiler provides a cross-reference table fcr all symbols.

A good rule for software designers to follow is: DO NOT BE SNEAKY. Software systems are generally written for use at more than one installation and will probably be maintained by many people. The programmer who writes code that is very difficult to understand makes the job of program maintenance very difficult. This point is also made by Brooks and Iverson (1969).

With standardization, as with virtually everything in computer software, it is difficult to define at what points trade-offs should be made. For example, how much time or space must be saved to justify writing code that modifies itself at execution time? In particular, if it is nearly impossible to understand what the programmer was trying to do, then the benefits derived are overshadowed by its lack of flexibility.

This is an important question for programmers who write low-level system software. Modules at this level are used repeatedly by the system, so efficiency is very important. However, it is vital that the system programmers maintaining the system can understand what the module does and how it works.

(3) Operating System Independence

An operating system-independent module is a module that does not require any of the services provided by the operating system. For example, the module contains no input/output requests or requests for storage.

Whenever possible, programmers attempt to make as many modules as possible independent of the operating system. Hence, if the software is to be used on another operating system, many modules will not require changes. For example, a language processor such as a FORTRAN compiler should be designed with as many modules as possible completely operating system-independent. Only modules such as input/output modules will then require changes before the compiler

can be used under ancther operating system.

With a well-defined interface and functional specification for a module which is to be changed, it is a relatively simple matter to change the compiler so that it functions under a different operating system.

(4) Program Documentation

Documentation will be discussed more completely in a later section. Regardless of the method of documentation chosen, it should be consistent throughout the software. Consistency of structure and style will allow the programmers who are working with the system to find information quickly, since documentation structure is the same for all parts of the software.

## 2.3 Efficiency and Stability

The cost of using software is a very important consideration in any software development project. The designer and programmer should attempt to design and write software that is as compact and as fast as possible. Software also must be stable. Stable software is software that has most 'bugs' removed and is not likely to 'crash' or 'hang-up' if it receives erroneous input. Rather than

'crashing', explanatory messages should be given and the error handled in an appropriate manner, such as asking for new input. Thus stability invclves checking for errors, which increases the cost of using the software. However, unstable software discourages users even more quickly than costly software. If the user makes an error, the system may respond with an explanatory diagnostic, an error recovery diagnostic, an uninformative message, a software crash, or, worst of all, erroneous results.

Frequently, large and/or complex software systems are programmed with two or more levels, as explained in the section on modularity. The highest level consists of the modules that are called by the user. The highest level modules give error diagnostics and possibly attempt error recovery. The needed stability is provided at this level. These routines, often without the user being aware of it, call subroutines at a lower level. For example, when a programmer codes a READ or WRITE statement, it appears to him that the FORTRAN compiler will generate the code necessary to perform the input or output. Actually, the compiler generates code to call the low-level input/output routines provided by the operating system. These lower level routines may have been written on the assumption that calls to them will be correct, that is, any erroneous data will have been detected before a call is made. If an error happens to be detected at this level, the messages are generally very uninformative, for example, 'SYSTEM ERROR'

and an error found at this level will probably cause the program to be abnormally terminated.

One advantage of having routines at more than one level is efficiency. The higher level routines are called by most users. However, the programmer who wants a particularly efficient operation will use the lower levels, even though his programming may be more involved. This allows programmers the opportunity to design efficient systems built on another system. An example of this type of hierarchical structure in IBM'S Cperating System/360 (1970-5) is the Queued Sequential Access Method (QSAM), which is a level above the Execute Channel Program routines (EXCP), that is, the QSAM routines make use of the EXCP routines.

The more services that are provided by the software, the greater the cost will be to the user of the software. Tasks such as providing error checking diagnostics and debugging aids involve considerable overhead. Because of this overhead, most large software systems should provide the user with a multi-level structure, such as the structure mentioned above, where the user can, if he wants, obtain efficiency at the cost of stability with slightly more programming effort.

## 2.4 Flexibility and Extensibility

Software packages should allow the user to perform some set of actions efficiently and with a minimum of effort. In particular, software packages should be user-oriented. A software system that makes it difficult for the user to do what he wants will not be used.

A great deal of effort is spent on the implementation of software packages. This effort can to some degree be wasted, if insufficient thought is given to the design of the software before implementation begins. Software that binds the user to a very limited set of operations that can not be extended will last only a short time, and then be replaced by something that may be only slightly better. This quick replacement of software packages, because of a lack of flexibility and extensibility, is very expensive and wasteful. Overall, it would be cheaper to consider more thoroughly the needs of the user, and design software that is flexible and extensible.

Software packages that are too general-purpose tend to become inefficient. As more options, alternatives, and features are added to a system, more checking and decision making is required to decide what is to be done. A well-designed software package should be no more general than necessary, but be flexible and extensible enough for changes and additions to be made to meet most future needs.

This idea has been expressed by Lampson (1970). 'If a system is to evolve to meet changing requirements, and if it is to be flexible enough to permit modularization without serious loss of efficiency, it must have a basic structure that allows extensions not only from a basic system but also from some complex configuration that has been reached by several prior stages of evolution. In other words, the extension process must not exhaust the facilities required for further extensions. The system must be completely open-ended, so that additional machinery can be attached at any point.' One may not always foresee all possible uses for a software package. It is possible to implement data structures that are efficient and open to extension. An example of this type of structure is given in Section 3.4. This type of design will allow software packages to last rather than have only a short life.

## 2.5 Documentation

Three basic pieces of documentation are needed to describe a software project. The first is the software specification written before the software is coded, and which is used as a guideline for implementation of the software. The remaining two documents, the Internal Logic Manual and the User's Guide, describe the software after development.

### 2.5.1 Software Specification

There are many different methods proposed in the literature for writing software specifications : 'How To Write Software Specifications' (Hartman And Owens 1967), 'A Guide For Software Documentation' (Walsh 1969), and 'Documentation Standards' (Gray And London 1970) Are Examples. The methods vary greatly. However, most are oriented towards the design cf large-scale software by manufacturers. It is not possible to state that one method is better than another, for needs vary greatly between installations. The intention here is to emphasize the importance and value of an adequate software specification. There is a tendency to begin programming without the initial step of writing a software specification because of the amount of work involved. Neglecting this step is generally more costly in the long run, for problems which were not foreseen may require the recoding of large sections of the software. If the implementor has a clear, precise, well-explained document before him frcm which he may code the software, the implementation will proceed much more rapidly and with less likelihood of running into unexpected prcblems.

One possible way to design and organize a software specification is given below. Fixed, rigid rules for writing a specification are not given. Instead, a few basic principles are given that can be applied by the designer in a way that best suits the situation.

Initially, after having given some thought to the problem, the designer should attempt to formulate a method for the storage and organizaticn of the information that will be used and manipulated by the software system. This method of organization of the data is commonly called a data structure. If the problem or system being programmed is fairly large, it can probably be broken down into several distinct parts, each of which will have its own data structure(s). Obviously, if the problem is large, it is likely that the first attempt at organizing the data structure will not be adequate.

By breaking the system down into logically separate components, the designer may begin to develop a modular system. By taking one logical unit at a time and considering the interaction of this unit with the data structure, inadequacies in the data structure may begin to appear. These may result in the formulation of a new data structure. This time, the designer has more insight into his problem and may design a better data structure. It is important to keep a record of changes, so that at any time one may get an accurate picture of the data structure and the functional relationships of the modules that so far have been proposed. The description of the functional relationship of the module to the data base is commonly referred to as the functional specification.

The way in which a written record is kept of the data structure(s) and functional specifications will vary greatly

from person to person and possibly problem to problem. Some people may prefer diagrams and flowcharts, while others prefer written notes. Many different systems and forms have been suggested and produced for this type of documentation, and the designer must decide which method is best suited to his needs.

As this process of modifying and redefining the data structure(s) and defining of functional specification for modules goes on, the designer may begin to see that the logical units proposed for the system are not a complete set, or not as clearly defined as was first thought. This may require the redefining of some modules, but eventually the system will take shape on paper. If a thorough job has been done, the greatest part of the design work is complete.

The third part of the total specification is the interface specification for each module, that is, a specification of how modules will link together and exchange information. To a large degree, the way the interface is accomplished will be defined for the operating system (Appendix III). However, what information is to be passed must be rigidly defined.

To some degree, parts two and three above may overlap, since communication between modules may be performed through the changes made to a common data structure(s). The interface specification is then very much a part of the functional specification.

In summary, three parts are required for a software specification:

1. Definition of data structure(s).
2. Functional specifications for each module.
3. Interface specification between modules.

The importance of an adequate and complete specification for the implementation of software cannot be over-emphasized. If the programmer is presented with a complete and detailed definition of the system, implementation can be realized quickly with little likelihood of unexpected problems.

PART I

PREFACE . . . . . (COMMON SECTIONS)

EFFECTIVE USE OF THIS MANUAL. . . . . 42

TABLE OF CONTENTS . . . . . (COMMON SECTIONS)

INTRODUCTION. . . . . 42

PART I (*First Program*). . . . . 43

SECTION A: PROGRAM LOGIC . . . . . 43

SECTION B: ROUTINE/SUBROUTINE LOGIC. . . . . 43

I. (*Routine/Subroutine Name*). . . . . 44

a) PURPOSE. . . . . 44
b) ENTRY POINTS . . . . . 44
c) EXIT POINTS. . . . . 45
d) TABLES REFERENCED. . . . . 45
e) PROCESSING PERFORMED . . . . . 46

*Iterate for Each Routine/Subroutine.*

SECTION C: PROGRAM LISTING . . . . . 46

APPENDIX: MAINTENANCE OF (*First Program*) . . . . . 46

GLOSSARY OF TERMS . . . . . (COMMON SECTIONS)

*PART I might be trivial in the cases where only one program is being described. In such cases the heading by PART I, etc. is not used. In cases where the manual discusses a number of programs which constitute a system, Sections A, B, C and the Appendix are repeated for each.*

FIGURE 2-2 Internal Logic Manual Index

C28-6534
C28-6535
C28-6646
A22-6821

Introduction to Control Program Logic
Y28-6605

MVT Control Program Logic Summary
Y28-6658

Catalog Management
Y28-6606

Sequential Access Methods
Y28-6604

C28-6628
C28-6650

On-Line Test Executive Program
Y28-6651

C27-6929
C27-6912
C27-6927
C28-6528

Graphic Problem Oriented Routines
Y27-7112

Graphic Access Method
Y27-7113

System 360 Data Processing from a Remote 1130 2250 Location
Y27-7156

C27-6912
C27-6931

Graphic Programming Services for FORTRAN IV
Y27-7152

Graphic Job Processor
Y27-7159

GRAPHICS

Indexed Sequential Access Methods
Y28-6618

C28-6586

Utilities
Y28-6614

C28-6514
C28-6538
C28-6647

TESTRAN
Y28-6611

C28-6603
C28-6629

FORTRAN IV (E)
Y28-6601

C28-6515
C28-6639

FORTRAN IV (G)
Y28-6638

C28-6515
C28-6602

FORTRAN IV (H)
Y28-6642

C27-6937
C28-6514
C28-5927
C27-2004

System/360-1130 Data Transmission for FORTRAN
Y27-7161

FORTRAN

C28-6514
C28-6538

Linkage Editor (E)
Y28-6610

C28-6514
C28-6538

C28-6551
C28-6628

Linkage Editor (F)
Y28-6667

C28-6514
C28-6535
C28-6539

Loader
Y28-6714

C28-6539
C28-6540
C28-6628

Job Management
Y28-6613

### 2.5.2 Internal Logic Documentation

Once the software is completed it is essential that its internal logic be documented. Maintenance can be a tiring and unrewarding job, and the job made even more tedious, perhaps even impossible, if the software is not adequately documented.

Again, as with the software specification, many methods have been suggested for the construction of internal logic documentation. Walsh (1969) and Gray and London (1970) provide examples of various methods for internal logic documentation. In general, the methods suggested for documentation tend to be very involved and require substantial resources to produce and maintain. This is generally not acceptable if the software being produced is the work of a small group with very limited resources.

Walsh (1969) offers a format in her book 'A Guide For Software Documentation' that appears to be very complete and easy to adapt to special needs. Figure 2-2 is an example taken from her book.

Regardless of the form chosen, there are several important pieces that together form complete internal logic documentation. First, there are the actual listings of the system which should be well 'commented', and organized in a manner that allows any module to be found rapidly. Second, there is the written documentation which gives an overview of the total system as a collection of interacting 'black-

boxes' as well as a detailed description of each 'black-box' or module. The written documentation should also include a detailed explanation of the tables and data structures used in the system. Third, there should be flowcharts describing the whole system in general and each module in detail. Finally, there should be a maintenance sheet so that changes to the system can be recorded. Section 3.5.2 and Appendix IV give examples of this type of documentation for GRIDSUB.

An example of complete internal program logic documentation can be seen in the way IBM has documented the software for Operating System /360 with publications called Program Logic Manuals or PLMs (IBM 1970-3). A view of the completeness of this documentation can be seen in Figure 2-3. Listings for the system modules are available on microfiche and included as part of the entire system documentation.

### 2.5.3 User's Guide

As with the other forms of documentation, there are many methods for documenting a system so that the user may understand the system and use it effectively. The User's Guide is quite distinct from the internal logic documentation, whose main function is to describe how the system works. The typical user does not require this knowledge, and the User's Guide is intended for people wishing to learn only how to use the system. The internal logic documentation is intended primarily for the maintenance personnel.

Generally, user's guides are written by someone who has developed or helped to implement the software.

It is difficult to say what user's guides should and should not contain. The level of expertise of users may vary greatly, from novice programmers to experienced systems programmers. It is important that this documentation be complete enough so that, with the User's Guide alone, the system may be used by the novice. It is the content, not the style, that is most important.

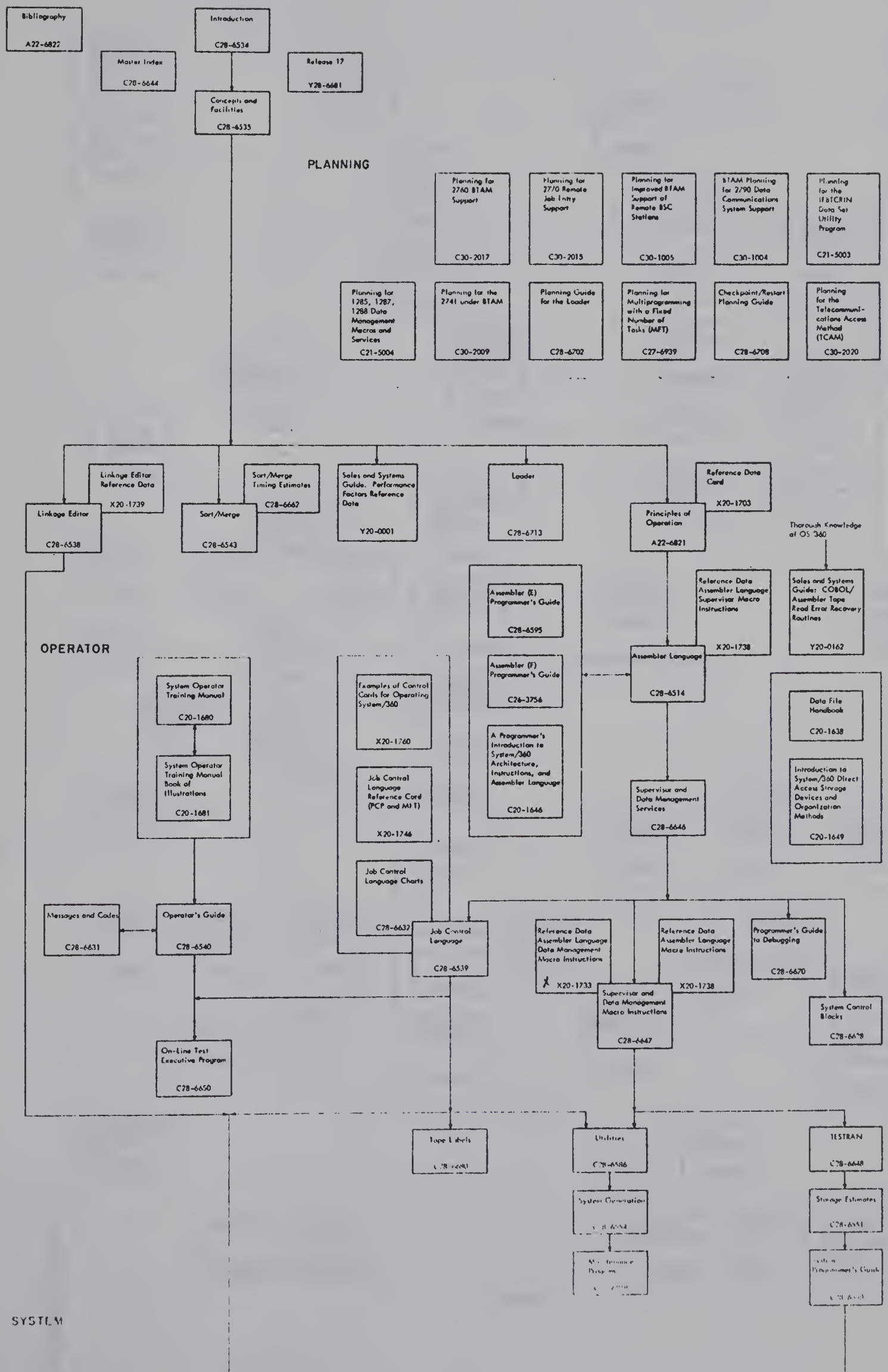
Again, some of the best and most complete examples of user documentation are the publications produced by IBM for Operating System /360 (IBM 1970-3). These publications are grouped under the general heading of System Reference Library Publications (SRLs) and are distinct from the PLMs. Figure 2-4 gives an overview of the SRLs for Operating

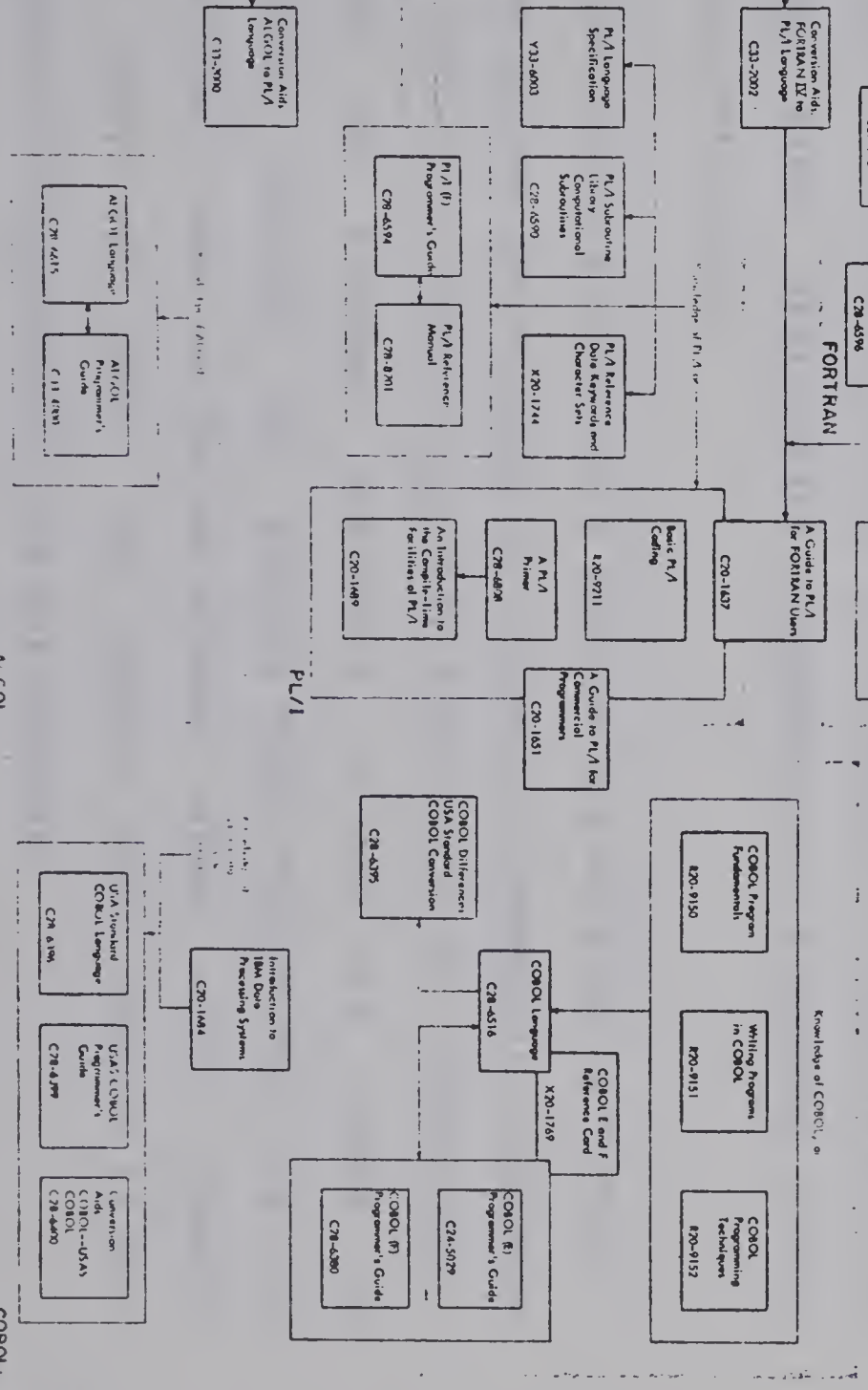
System/360. The SRLs are intended to be general information for users, whereas the PLMs are intended for maintenance perscnnel.

An important requirement for all three forms of documentation is that updating be systematic. As the system evolves and is developed or expanded, it is important to document the changes, so that the documentation describes the system as it exists at the present time. If documentation is well-organized in distinct, well-defined sections, changes to the system may require changes only to isolated sections of the documentation. One possible solution is the use of text processing packages to maintain and update the documentation. Packages such as TEXT/360 (IBM 1969-6) and FMS (Silver 1969) are very convenient for maintaining documentation.

As changes occur, updates may be machine produced to replace the changed sections in the documentation. Periodically, a new version of the manual may be produced which will incorporate all previous updates.

The major advantage of such an updating system is the ease with which changes may be made and new copies or updates produced. The major disadvantage of such a system is the cost of producing many copies. However, if one master copy is produced, the remaining copies may be produced from this master using more conventional means which are more economical.


Bibliography
A22-6822
Master Index
C28-6644
Introduction
C28-6534
Release 17
Y28-6681
Concepts and Facilities
C28-6535
PLANNING
Planning for 2760 BTAM Support
C30-2017
Planning for 2770 Remote Job Entry Support
C30-2015
Planning for Improved BTAM Support of Remote BSC Stations
C30-1005
BTAM Planning for 2790 Data Communications System Support
C30-1004
Planning for the IFBTCRIN Data Set Utility Program
C21-5003
Planning for 1285, 1287, 1288 Data Management Macros and Services
C21-5004
Planning for the 2741 under BTAM
C30-2009
Planning Guide for the Loader
C28-6702
Planning for Multiprogramming with a Fixed Number of Tasks (MFT)
C27-6939
Checkpoint/Restart Planning Guide
C28-6708
Planning for the Telecommunications Access Method (TCAM)
C30-2020
Linkage Editor
C28-6538
Linkage Editor Reference Data
X20-1739
Sort/Merge
C28-6543
Sort/Merge Timing Estimates
C28-6662
Sales and Systems Guide. Performance Factors Reference Data
Y20-0001
Loader
C28-6713
Principles of Operation
A22-6821
Reference Data Card
X20-1703
Thorough Knowledge of OS 360
Assembler (E) Programmer's Guide
C28-6595
Assembler (F) Programmer's Guide
C26-3756
A Programmer's Introduction to System/360 Architecture, Instructions, and Assembler Language
C20-1646
Assembler Language
C28-6514
Reference Data Assembler Language Supervisor Macro Instructions
X20-1738
Sales and Systems Guide: COBOL/ Assembler Tape Read Error Recovery Routines
Y20-0162
OPERATOR
System Operator Training Manual
C20-1680
System Operator Training Manual Book of Illustrations
C20-1681
Examples of Control Cards for Operating System/360
X20-1760
Job Control Language Reference Card (PCP and MFT)
X20-1746
Job Control Language Charts
C28-6637
Supervisor and Data Management Services
C28-6646
Data File Handbook
C20-1638
Introduction to System/360 Direct Access Storage Devices and Organization Methods
C20-1649
Messages and Codes
C28-6631
Operator's Guide
C28-6540
Job Control Language
C28-6539
Reference Data Assembler Language Data Management Macro Instructions
X20-1733
Reference Data Assembler Language Macro Instructions
X20-1738
Programmer's Guide to Debugging
C28-6670
Supervisor and Data Management Macro Instructions
C28-6647
System Control Blocks
C28-6678
On-Line Test Executive Program
C28-6650
Tape Labels
Utilities
C28-6586
IESTRAN
C28-6648
Storage Estimates
C28-6551
SYSTEM


Graphic Programming Services for FORTRAN IV
C27-6932
FORTRAN IV (G&H) Programmer's Guide
C28-6817
1130/1800 Basic FORTRAN IV Language
C26-3715
System/360 and 1130 Disk Monitor System Data Transmission for FORTRAN
C27-6937
Conversion Aids: FORTRAN IV to PL/I Language
C33-2002
FORTRAN IV Language
C28-6515
FORTRAN IV Reference Card
X28-6383
Model 44 Programming Support Assembler Language
C28-6811
FORTRAN IV Library Subprograms
C28-6596
Basic FORTRAN IV Language
C28-6629
FORTRAN
PL/I Language Specification
Y33-6003
PL/I Subroutine Library Computational Subroutines
C28-6599
PL/I (F) Programmer's Guide
C28-6594
PL/I Reference Manual
C28-8201
PL/I Reference Data Keywords and Character Sets
X20-1744
Conversion Aids: ALGOL to PL/I Language
C33-2000
ALGOL Language
C28-6615
ALGOL Programmer's Guide
C28-6594
A Guide to PL/I for FORTRAN Users
C20-1637
Basic PL/I Coding
R20-9211
A PL/I Primer
C28-6808
An Introduction to the Compile-Time Facilities of PL/I
C20-1689
PL/I
A Guide to PL/I for Commercial Programmers
C20-1651
Introduction to IBM Data Processing Systems
C20-1684
COBOL Differences: USA Standard COBOL Conversion
C28-6395
USA Standard COBOL Language
C28-6396
USA Standard COBOL Programmer's Guide
C28-6397
COBOL Program Fundamentals
R20-9150
Writing Programs in COBOL
R20-9151
COBOL Programming Techniques
R20-9152
COBOL Language
C28-6516
COBOL E and F Reference Card
X20-1769
COBOL (E) Programmer's Guide
C24-5029
COBOL (F) Programmer's Guide
C28-6380
Conversion Aids: COBOL--USAS COBOL
C28-6470
COBOL
Knowledge of COBOL, or
Graphic Programming Services for 2780 and 2782 Film Units
C27-6937
Basic Programming Support FORTRAN IV (Tape) Programmer's Guide
C28-6583
Basic FORTRAN IV (E) Programmer's Guide
C28-6603
Disk and Tape Operating System FORTRAN IV Programmer's Guide
C24-5038
GRAPHICS
Graphic Programming Services for FORTRAN IV
C27-6913
User's Guide for Job Control from the 7250 Display Unit
C27-6931
Component Description: 7250 Display Unit Model 1
A27-2701
Component Description: 7250 Display Unit Model 2; 2848 Display Control
A27-2711
Component Description: 7250 Display Station, 2848 Display Control
A27-2700
Graphic Programming Services for 2260 Display Station (Local Attachment)
C27-6912
Component Description: 7250 Display Unit Model 4
A27-2723
User's Guide for Job Control from 7250 Display Units Attached to an 1130 System
C27-6938
OLIAM Message Processing Program Services
C30-2003
Message Control System Concepts
C20-1699
Data Communications Primer
C20-1648
Component Description: 2703 Transmission Control
A27-2703
1030 Data Collection System
A24-3018
1050 Data Communication System Principles of Operation
A24-3474
2701 Data Adapter Unit Principles of Operation
A27-6864
2703 Transmission Control
A27-6846
Component Description: 2740 Display Station, 2848 Display Control
A27-2700
2740 Communications Terminal
A24-3403
OLIAM Message Control Program
C30-2005
BTAM
C30-2004
Remote Job Entry
C30-2006
TELECOMMUNICATIONS

# CHAPTER III

# SOFTWARE DESIGN PRINCIPLES ILLUSTRATED IN A GRAPHICAL SUBRCUTINE PACKAGE

## 3.0 Introduction

GRIDSUB, a Graphical Subroutine Package, was originally proposed and written by W.H. Huen (1969). The first package was designed to operate on a Control Data Corporation GRID (Graphical Remote Interactive Display) Graphic Terminal Subsystem (CDC 1969) with one bank (4096-12 bit words) of core. Extension of the GRID hardware to three banks made the old software obsolete.

A new GRIDSUB package has been written by the author to operate on the extended GRID hardware. This new package was written to be compatible with the old GRIDSUB for most applications programs, except that software capabilities for handling large application programs were increased by the hardware extension.

The original subroutine package was written in FORTRAN. The new system was written in System/360 Assembler Language (IBM 1970-2) to achieve greater efficiency and compactness. Although the internal structure of the new system is completely changed, any program which worked with the cld system should work on the new system, unless nested blocks (Huen 1969) are used. The nested block feature is not available at this time in the new GRIDSUB.

An objective in re-writing GRIDSUB was to design the software package in a manner which demonstrates the general principles of software design discussed in Chapter II. This chapter relates the principles discussed in Chapter II to the actual implementation and design of the new GRIDSUB.

## 3.1 Modular Construction Of GRIDSUB

The two previous definitions of modularity, given in Chapter II, both apply to GRIDSUB modules. Each GRIDSUB routine is a closed subroutine that is not stored in the main path of another routine and is entered by a jump operation. Every GRIDSUB routine is a program unit that may form the input to or output from a single execution of the assembler and is discrete and identifiable with respect to combining with other program units and loading.

The linkages between all subroutines of GRIDSUB are the standard calling sequences defined for IBM software. This allows all routines to be called from FORTRAN routines and facilitates ease of communication within GRIDSUB.

Examples of a module's functional specification are included in the section dealing with GRIDSUB documentation.

The set of GRIDSUB modules, which are explained in Appendix I, may be classified into four categories; each category containing modules for a set of related functions.

| Block Definition | | |
|---|---|---|
| BLOCK | ENDBLK | PCINTX |

MOVE VECTOR TEXT

DLINE IDY

Display Definition and Transmission

DISPLY TRANMT

Message Decoding

DLPEN DALPHA DFKEY

DVECT DPOINT DEND

Display Modification

MOVEBK DELBLK FREEBK

ERSEK DELID RSTBK

The above four categories may also be considered to be the four types of graphical functions performed by an interactive graphics program using GRIDSUB.

In a typical case, a user's program will initially call the block definition routines to define the basic pictoral entities to be displayed. The program will then call the display definition and transmission routines to define his picture and send it to GRID for display. Upon return from the transmission routine, the user has available to him the message created at the graphics terminal by the operator. The program will interpret this message with the aid of the decoding routines. Based on the interpretation of the decoded message, the program will modify the picture using the display modification routines. Definition of new elements to be added to the picture will result in the program again calling the block definition routines.

Essentially, an interactive graphics program is an interpreter of messages comprised of actions (light pen picks, pressing of function keys) performed by the console operator.

The sequence just described is a logical ordering that describes the structure of most interactive graphics programs using GRIDSUB.

Using the four categories of routines defined above, it should be possible to reduce the paging requirements of most application programs by loading those GRIDSUB routines in one category on the same page or group of pages. Thus, block definitions can be performed with a minimum number of pages required in core. This reduces paging overhead by requiring the user to reference as few pages as possible. The likelihood of all these pages remaining in core for the duration of the block definitions is much higher than if the block definition routines were spread throughout most pages of the GRIDSUB system.

As an application-oriented package becomes larger and more diversified the application package is often made available to the user as a library in a single data set that contains modules and a directory which identifies and locates the modules. The advantage of a library is that the user's program has loaded with it only those routines to which it refers either directly or indirectly.

In any environment where the user is charged for the

memory space that his program requires, a library can provide a saving if the user is not using all the routines stored in the library.

Under the M.T.S. system, the charge is computed by multiplying the amount of CPU time used by the amount of virtual memory in use. Since the amount of memory in use can vary at execution time, a running sum is kept for this charge. This is referred to as the M.T.S. Virtual Memory Integral. With this method, the charge for virtual memory increases as the use of CPU time increases. For this reason, it becomes important to reduce the virtual memory requirements where possible. Therefore, GRIDSUB was made available to the application programmer as a library, rather than as a single load module.

A charge for memory requirements is typical of most systems. For this reason, a library may be the most economical method for making a system available. However, the cost of searching the directory and the saving in space should be weighed carefully before a decision is made as to which is the best method.

Graphical application programs tend to have a relatively high 'in-core' time with a relatively low proportion of CPU time used, due to the operator interaction at the graphics terminal. The operator requests some picture-modifying action and then studies the result. Based on what he sees, a new command is given. This results in the low proportion of CPU time utilized by the program,

usually 1% to 10%. For this reason, if a memory charge is assessed on the basis of elapsed time, reductien cf memory requirements becomes extremely important if interactive graphics is tc be economical. In a multiprogramming system, the use of a program library may be the most effective way to reduce virtual memory charges for the typical user, who is using only a subset of the total system.

## 3.2 Standardization in GRIDSUB

The designers of a system usually do not maintain the system once it has been completed and is in use. The task of maintenance often falls to people who have not had any previous contact with the software. To aid maintenance personnel, designers should attempt to write the software in a consistent manner, observing as many standard practices as possible.

During the design of GRIDSUB, attempts were made to follow the fcllowing standard software practices:

(1) Linkage Conventions

The GRIDSUB system was designed to be available under two operating systems, OS/360 and M.T.S. Both operating systems use the standard CS type calling sequence (IBM 1970-5) shown in Appendix III.

The standard linkage conventions make

inter-program communication and module replacement very straight-forward. A valuable debugging aid is provided by a record of register contents and a trace of the routines through which control has passed shown by means of the forward and backward pointers in the save areas (Figure 3.1).

(2) Programming Practices

Throughout GRIDSUB, register numbers have been equated (IBM 1970-2) to symbolic names. GRIDSUB is assembled using the M.T.S. (G) Assembler (University of Alberta 1970) which provides a complete cross-reference table for all symbols used in an assembly. This is just one example of what was defined earlier (Chapter II) as a good programming practice.

Within GRIDSUB, symbolic names have been used as labels for branch instructions rather than branching by using displacements. For example:

```
B    START     BRANCH TO START
```

is used rather than:

```
B    *+30      BRANCH 30 BYTES FORWARD
```

This method, using symbolic names for

labels, allows the programmer to find quickly all references to a label in the cross-reference table created by the assembler. Use of symbolic names also allows a programmer to insert code between a branch instruction and the code being branched to without having to change the branch instruction.

The code in GRIDSUB has been written in a straight-forward manner. Instructions are individually commented. Nowhere in GRIDSUB is code modified at execution time. This approach does add slightly to the cost of using GRIDSUB. However, it is much easier for anyone working with GRIDSUB to debug, modify, and/or add code to GRIDSUB routines.

(3) Operating System Independence

Wherever possible GRIDSUB was divided into modules which performed one or more logical functions. Since the intention was to have the package available under more than one operating system, most of the modules are operating system-independent. GRIDSUB must function under both M.T.S. and OS/360. The major change needed to convert GRIDSUB from one operating system to the other was in the

input/output, both for error diagnostics to the user and for GRID to System/360 communication. For this reason, GRIDSUB was set up to be assembled either for M.T.S. or OS/360 from one source deck which contained conditional assemblies (IBM 1970-2). For example, the following statements in the source program allow the programmer to vary the way in which GRIDSUB is to be assembled by varying the file SYSTEM:

```
    LCLC    &SYSTEM
    COPY    SYSTEM
```

This file contains a declaration of the conditional assembly variable, &SYSTEM, stating for which operating system GRIDSUB is to be assembled. The COPY assembler command (IBM 1970-2) copies the file SYSTEM into the source program that is to be assembled. For M.T.S., the file SYSTEM should contain:

```
    &SYSTEM   SETC      'MTS'
```

For OS/360 the partitioned data set member SYSTEM should contain:

```
    &SYSTEM   SETC      'OS'
```

By changing the file SYSTEM, a module which was not cperating system-independent could be assembled for either OS/360 or M.T.S., and only one source deck would be required. Having only cne source deck for a

given module makes updating of GRIDSUB routines much easier, since there is only one change to make rather than a change to two decks. This virtually assures exact compatability between operating systems, since the only part of the system that is different is the part which is operating system-dependent. This was accomplished using conditional assembly statements such as those shown below:

```
            AIF     ('&SYSTEM NE 'OS').MTS
            PUT     MESSAGE,ERROR
.MTS        AIF     ('&SYSTEM' NE 'MTS').END
            WRITE 6,ERROR,121
.END        ANCP
```

With the above conditional assembly statements, the PUT macro would be assembled if &SYSTEM was set to 'OS'. However, if SYSTEM was set to 'MTS', then the WRITE macro would be assembled.

The use of conditional assembly statements was restricted to modules which were not completely operating system-independent, and where any attempt to make them operating system-independent would destroy their logical completeness.

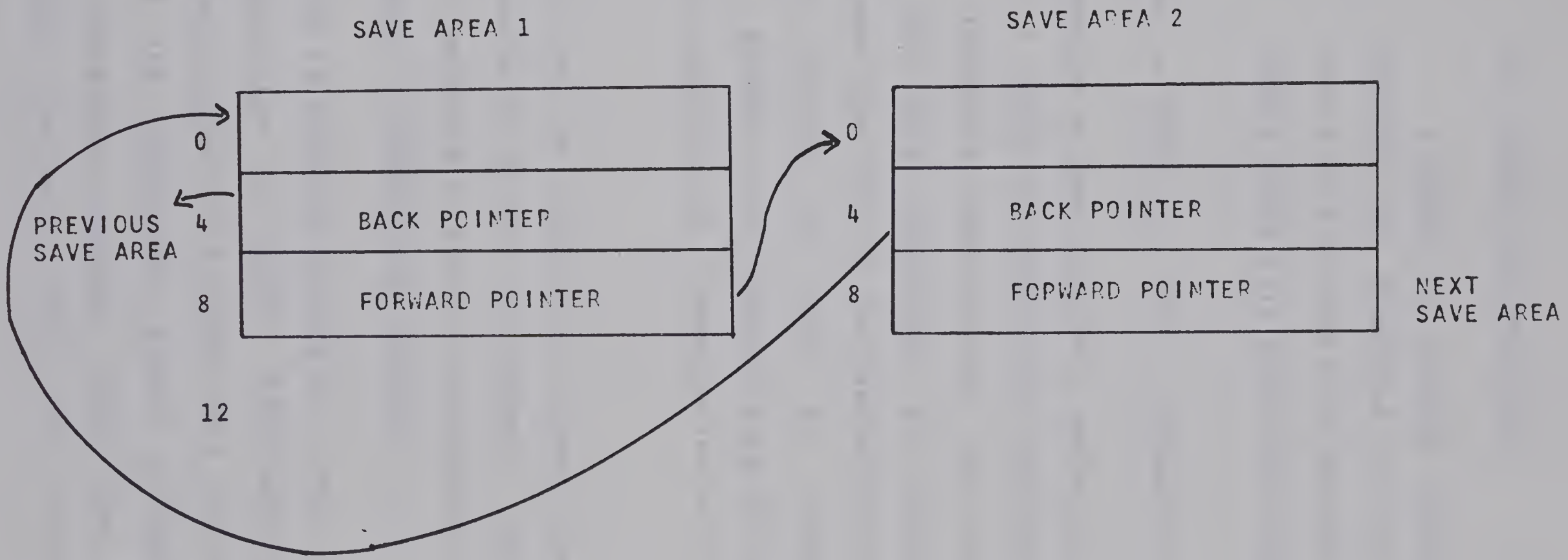



FIGURE 3-1 SAVE AREA LINKAGE

(4) Program Documentation

A method of program documentation for GRIDSUB was described in this thesis. The method is illustrated in Section 3.5 of the thesis which will deal with documentation.

## 3.3 Efficiency vs Stability, GRID - System/360 Communication

In attempting to make GRIDSUB a system of logical units or modules, a group of routines was designed to deal only with GRID - SYSTEM/360 communication. One routine, WRTGRID, writes to GRID from System/360, and another routine, READGRID, reads from GRID at the System/360 end. There are complementary routines in GRID to communicate with System/360. WRTGRID and READGRID form the lower level of the two level structure for communication at the System/360 end.

To the user of GRIDSUB, it appears that TRANMT only prepares messages to be sent or received. The sending and receiving of messages is actually controlled by the channel programs issued by WRTGRID and READGRID.

Stability is achieved through the error checking and message preparation in TRANMT. Efficiency is achieved by having special-purpose I/C routines that perform no error checking on the information passed to them from TRANMT, since this information, assuming TRANMT has no bugs, is always correct. Any errors which occur at this lower level

appear to the I/O routines as transmission errors.

The I/O routines do contain error recovery. This error recovery is for transmission errors, not for logic errors due to erroneous parameters from TRANMT. For this reason, errors at this level produce error messages indicating unit checks and unit exceptions (IBM 1969-1). The corresponding error diagnostic messages tend to be very uninformative, for example: 'UNIT CHECK IN 2701 - SENSE BYTE = 01'. Not only is this message uninformative to the typical user, it is also misleading if the error was actually a result of bad parameters passed to the I/O routines. The case of erroneous parameters being passed from TRANMT to the I/O routines should never happen once the system is debugged. However, the user can expect this type of message if he bypasses TRANMT for the sake of efficiency, and attempts to program at this lower level.

The major advantages of this type of two level structure come from having a set of I/O routines that are not strictly tied to use in the GRIDSUB system, but are extremely special-purpose and efficient for use in the GRIDSUB system. These routines are very similar to what is normally called device-support routines.

The user must determine if the additional effort required to use the I/O routines directly will give a sufficient saving of time and space to warrant not using the general-purpose routine TRANMT, which provides the error

checking and message preparation that gives the inter-machine communication stability.

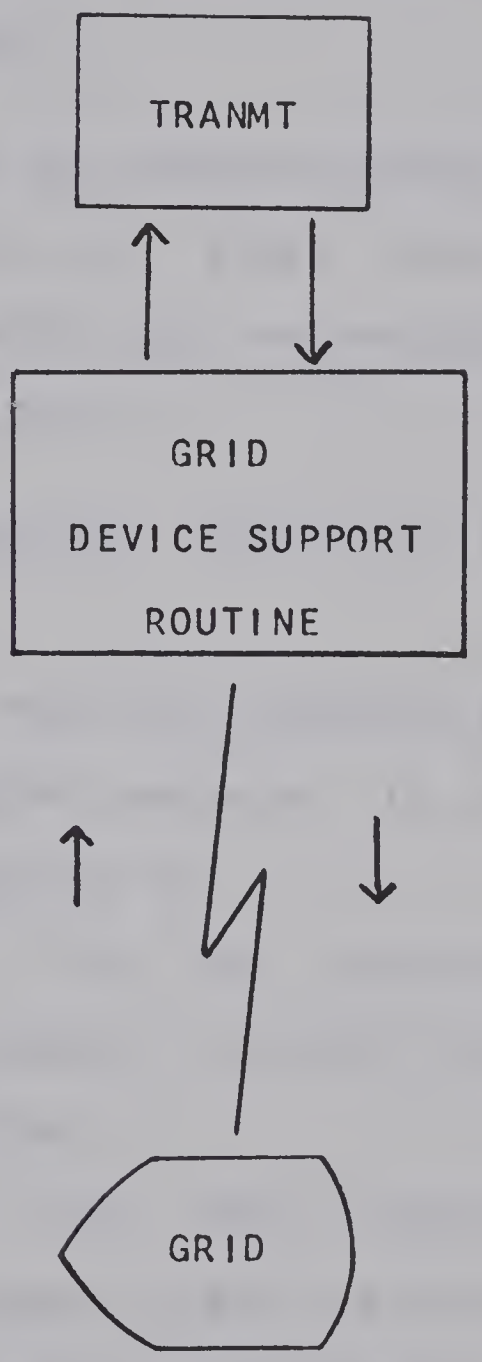



Figure 3-2 A Lower Level In Communication

## 3.4 Flexibility and Extensibility in GRIDSUB

The new GRIDSUB package was designed to appear, to the user, identical with the original GRIDSUB. Although the internal structure was changed considerably, the user who interfaced with GRIDSUB through the subroutines provided should see no change.

One example of the changed internal structure, the form of the Block File, is given below to show how the new GRIDSUB uses a flexible and extensible data structure to store blocks (Appendix I).

The data structure described below was chosen for several reasons:

(1) The data structure meets the needs of the GRIDSUB system as it is currently defined (Appendix I).

(2) The data structure is designed for efficient, straight-forward garbage collection.

(3) The data structure can be readily expanded to meet an individual user's needs.

(4) The data structure uses all available space before garbage collection is required.

(5) The data structure should be able to handle most expansions to the GRIDSUB system, if an expansion to the subroutine package is desired.

Four pointers or counters are maintained for the Block File (Figure 3-3):

BLKFILE -Block File address

BLKSP -Amount of space available within the Block File (initially the Block File size).

POINTER -Address for next pointer to code in Block File (initially the end of Block File)

CODE -Address of first available location for code to be inserted into the Block File (initially BLKFILE).

These pointers or counters are updated by the GRID code generation routines and by BLOCK and ENDBLK (Appendix I). The Block File is cleaned up by the garbage collector, GARBAGE, which resets the pointers or counters accordingly.

Figure 3-3 is a diagram of the Block File. GRID code is added from the bottom up. GARBAGE is called when the code meets the pointers (BLKSP=0).

A Block Location Table, BLT, of fixed size, is maintained to point to the pointers in the Block File. BLT is 512 four byte elements, corresponding to the 511 allowable blocks. Since Block 0 is not permitted, BLT(0) is not used as a BLT element. The BLT element address for a block is the BLT address plus the BLT displacement (four times the block number).

When the user begins a block definition with a call to subroutine BLOCK (Appendix I), the BIT element for that block is loaded with the address contained in PCINTER and the high order byte is set to zero. The BIT displacement is stored in the halfword pointed to by CODE, and CODE is incremented by two. The high-order byte of the address contained in POINTER is set to zero, and the address contained in POINTER is decremented by four. BLKSP is decremented by six. CODE is stored at the address contained in POINTER and the high-order byte at this address is set to hexadecimal 01 to indicate the last pointer. As the user calls the code generation routines (Appendix I) to build a block, GRID code is inserted at the address contained in CODE, CODE is incremented and BLKSP is decremented. The location pointed to by the last Block File pointer is set to the address contained in CODE.

When a block is required, the address of the Block File pointer for that block is found by referencing the BLT with the BLT displacement for that block (block number times four). The desired block is the area bounded by the address contained in the Block File pointer for that block and the address contained in the pointer immediately above the Block File pointer for that block.

If a block definition is deleted by a call to DELBLK (Appendix I), the high order byte in the BIT element and in the Block File pointer for the block is flagged with hexadecimal 80 to indicate garbage. If the block is then

redefined, the BLT element for this block is updated accordingly, but the old pointer is not changed. A new Block File pointer is established. The pointers are kept in order and are only changed when garbage collection is performed by GARBAGE.

Garbage collection is performed when more space is required than is available in the Block File. BLKSP is the counter maintained to keep track of available space in the Block File. Garbage collection begins at the top of the Block File by testing the high order byte of the first Block File pointer that is flagged to indicate garbage. This search continues with the next pointer (1 word lower in the Block File). When garbage is encountered, the address is saved and the scan of the Block File pointers continues, now searching for a pointer that is not flagged. The area delimited by the first pointer found to be flagged and the next pointer which was not flagged is an area into which good code can be moved, replacing the garbage. Obsolete Block File pointers are updated. The BIT elements for blocks which were moved are updated using the BLT displacement stored in the first half-word of the block code as a displacement to locate the BLT element. The scan continues until all garbage has been deleted, then CODE, BLKSP and PCINTER are updated.

Space for the Block File is obtained at execution time and can be easily changed by modifying the size parameter, BLKSZ. This method allows the user to expand the Block File

if necessary. Since space is not allocated separately to Block File pointers and Block code, the data structure is suitable for all types of program, whether there are to be many small blocks or a few large blocks.

An example is shown below of how this structure can be expanded to encompass the nested block feature implemented by Huen (1969). Other expansions to the Block File could be made similarly to incorporate nested blocks.

The nested block feature allows the user of GRIDSUB to include a previously defined block within the block currently being defined. A nested block is indicated when a pointer in the Block File begins with hexadecimal 40. The address portion of this pointer indicates the halfword immediately following the code that so far comprises the block being defined. This halfword contains the BIT displacement for the block that is being nested. A block definition is considered complete when a pointer is encountered that begins with hexadecimal 00 or 01. Pointers that begin with hexadecimal 80 are used as delimiters to separate code that has been deleted from code that has not been deleted. 'Deleted' code is code that is no longer used. For example, if the definition of a block is deleted using subroutine DELBLK, then that code is deleted and it is possible to redefine that block. The major advantage of having nested blocks described in this manner is the saving in space obtained by having the code for a block, whether nested or not nested, stored only once in the Block File.

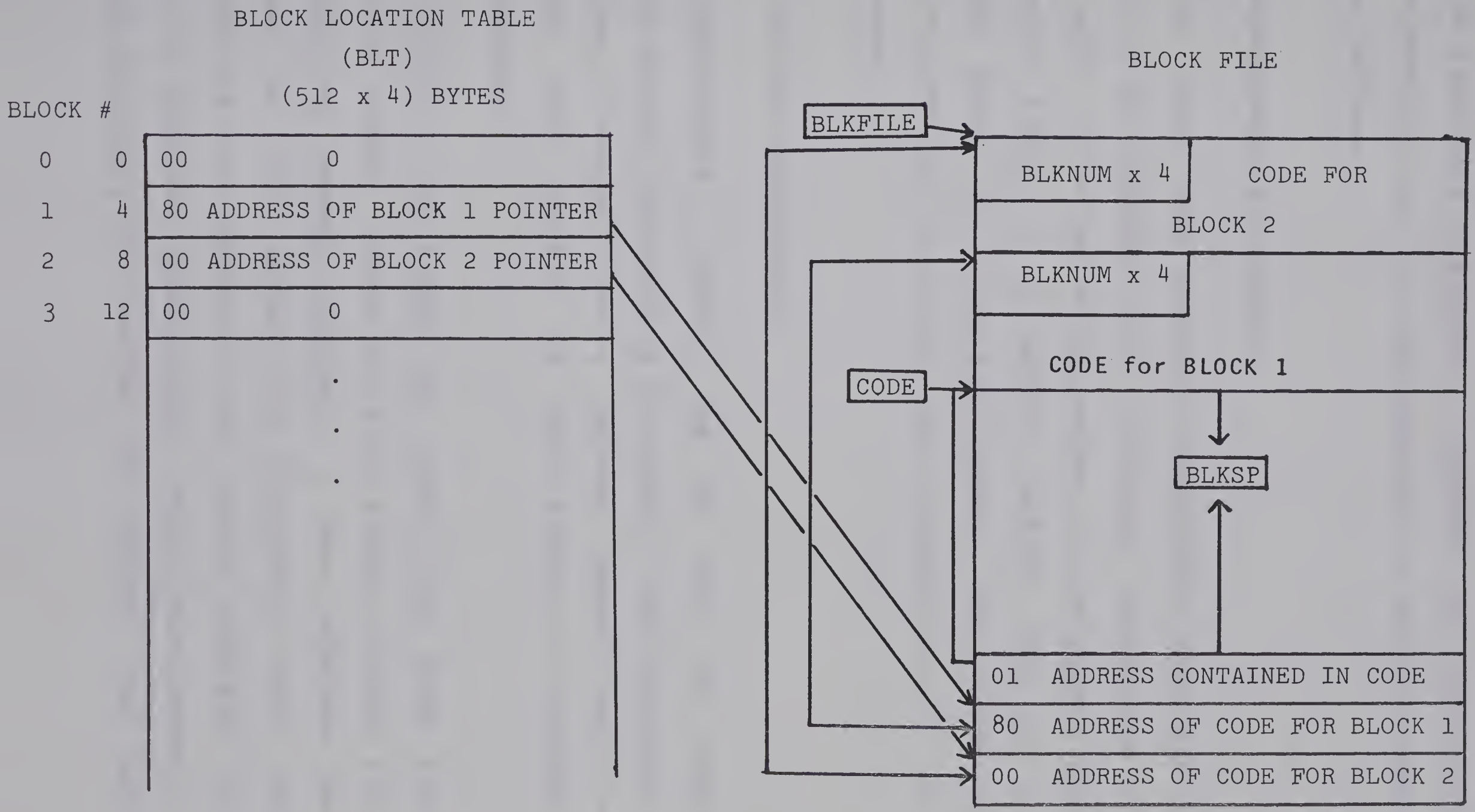



FIGURE 3-3 BLOCK FILE

The importance of adequate, flexible, and extensible data structures cannot be over-emphasized in the development of good software.

## 3.5 Documentation for GRIDSUB

The three basic pieces of documentation described in Chapter II are discussed in the following sub-sections as they apply to the new GRIDSUB system. At the present time the User's Manual is in machine readable form and is produced using a text formatting package. Work is about to begin on producing the internal logic documentation in the same manner.

### 3.5.1 Software Specification

Unfortunately, GRIDSUB was the first very large software project undertaken by the author. When the project was begun, it was felt that a finished product would be achieved more quickly by not writing a software specification.

Initially, the system was broken into three more manageable pieces. These were block storage, display file management and message decoding. A data structure was proposed for block storage and programming began. The procedure was very similar to the method suggested for writing a software specification, except that instead of verbally describing the modules for GRIDSUB, they were

programmed. As development progressed, deficiencies in the data structure became apparent and considerable reprogramming was required. In the previous section of this Chapter, a description of the Block File and a brief functional specification for several modules has been given. If this had been done before programming began, much recoding could have been avoided.

For example, the original format for the Block File structure was adequate for storing the blocks as defined by the user. However, the data structure was organized in such a way that garbage collection was impossible. Space made available by deleting blocks was not reusable. The user also could only define a maximum of one hundred blocks.

A functional specification should have been written for the Block File garbage collector. As a result, the data structure had to be redesigned and the programming redone.

The material presented in Section 3.4 of this thesis may be considered as one possible way to write a software specification. If this approach had been taken, system implementation would have been realized much sooner.

### 3.5.2 Internal Logic Documentation

GRIDSUB is a large software system, requiring about 12,000 source program cards. Due to this large size, the internal logic documentation for one module only of GRIDSUB is given (Appendix IV).

An overview of the GRIDSUB system is shown in Appendix V. The material in this Appendix is intended to be the 'black-box' description of the system mentioned in Section 2.5.2. This general description is intended to provide the necessary information about the system from which the more detailed description of individual modules may be approached with more insight.

Appendix IV is an example of the program logic documentation for a single module. The three basic pieces of information needed for complete internal logic documentation were mentioned in Section 2.5.2. These are:

1. Written documentation describing the module.
2. Flow charts for the module.
3. A source listing for the module.

The example given in Appendix IV is for a very small module, but even for such a small module a considerable amount of work is required for complete documentation. When planning a system, the work required for complete documentation should not be under-estimated.

### 3.5.3 User's Guide

Since GRIDSUB is a user-oriented software package, considerable effort was needed to prepare a suitable reference for the user. The User's Guide, Computer Graphics for the Application Programmer (Jackson 1972), is included in part within this thesis (Appendix I).

The Table of Contents for the User's Guide is shown in Figure 3-3. The manual is written in sections so that a change to the system should result in a change only to isolated sections within the documentation. This greatly simplifies the problem of maintaining 'current' documentation. At this time, the GRIDSUB User's Manual is produced with the aid of the text formatting package. Section 3 of the User's Guide is included as Appendix I to provide the reader with a sample of user documentation as well as a description of the GRIDSUB software.

The format of the table of contents (Figure 3-4) may be summarized as follows:

1. General Introduction
2. Hardware Description
3. Software Description
4. Operating System Interfacing
5. Examples
6. Appendices

These six parts are the basic components for this type of user's guides. Not all sections are applicable for all software. It is hoped that the six general headings can serve as a guide for the preparation of most user oriented documentation describing hardware/software packages similar to GRIDSUB.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

1. INTRODUCTION 1

2. THE GRAPHICS TERMINAL 2

2.1 GRID Description 2

2.2 GRID Supervisors 3

2.3 The MULTIBANK Supervisor 7

2.4 Display Terminal Behavicur under the MULTIBANK Supervisor 8

2.4.1 Light Pen Picks 8

2.4.2 Status/function Keys 9

2.4.3 Alphanumeric Keys 11

2.4.4 Other Keys 12

2.4.5 Display State 13

3. FORTRAN SUBROUTINES -- THE GRIDSUB PACKAGE 14

3.1 Introductory Description 14

3.2 GRIDSUB Specificaticns 16

3.2.1 Error Handling 16

3.2.2 Block Definition 16

(1) BLOCK 16

(2) ENDBIK 17

3.2.3 Graphics Element Generation 17

(1) MOVE 17

(2) VECTOR 17

(3) DLINE 18

Figure 3-3 GRIDSUB User's Guide Table of Contents

(4) POINTX 18

(5) TEXT 19

(6) IDY 20

3.2.4 Block Display

(1) DISPLY 20

3.2.5 Display Modification 21

(1) MOVEBK 21

(2) ERSBK 1

(3) RSTBK 21

(4) FREEBK 22

(5) DELID 22

(6) DELBIK 22

(7) LP 23

3.2.6 Tranmission and Decoding 23

(1) TRANMT 23

(2) DLPEN, DFKEY, DAIPHA, DVECT, DPOINT, DEND 23

3.2.7 Data Conversion 25

(1) CHRFLT 25

(2) INTCVT 26

(3) FLTCHR 26

(4) INTCHR 27

(5) DMPHEX 27

3.2.8 Specification of Logical Arguments 27

Figure 3-3 (ccntinued)

4. WRITING AN APPLICATIONS PROGRAM 29

4.1 Operating System Environment 29

4.1.1 OS/MVT 29

4.1.2 MTS 29

4.2 Overall Program Structure 29

4.3 Control Cards 30

4.3.1 OS/MVT 30

4.3.2 MTS 31

5. EXAMPLES

5.1 GRIDSUB examples 32

5.1.1 Block Definition 32

5.1.2 Transmission and Decoding 34

5.2 Complete Program

Appendices

I GRID Character Set 39

II OFFLINE-LOADER 40

III File and Table Space Usage under GRIDSUB 42

IV Display Command Execution Times 45

V MULTIBANK BOOTSTRAP 46

VI Screen Dump 47

VII Block Information 49

Figure 3-3 (continued)

# CHAPTER IV

# CONCLUSION

At the present time, the art of software design is not at a comparable level with other areas in computing science. The main reason for shortcomings in software design is the lack of generally accepted principles that may be used effectively in the construction of good software. The intention in this thesis has been to formulate and discuss several general design principles, and illustrate their importance by reference to a software package implemented by the author. In summary, the principles discussed in this thesis are briefly reviewed below.

Modularity is the dividing of a program into separate, clearly defined units that are more manageable to program and debug. Modularity makes possible the use of overlay structures and the creation of program libraries to allow more efficient use of resources. The clear definition of modules as parts of a total system readily facilitates group development in large software projects.

Standardization makes the job of program maintenance and program writing much simpler. By adhering to a standard linkage one module may be readily substituted for another. The programmer, by following standard practices such as the use of symbolic addressing and straightforward well commented code, can simplify the job of program maintenance

which may be the responsiblity of other programmers. By making as many modules as pcssible operating system-independent, an operating system change will require that cnly a minumum number of modules be modified. Standardization of program documentation can greatly simplify the problems of searching for information.

Efficiency and stability may be built into a system without one extensively interferring with the other by constructing a multi-level program structure. The modules at the upper levels can provide extensive error checking while the modules at the lower levels are special purpose routines that require little cr no error checking. The routines at this lower level can be made available for the more sophisticated user who seeks efficiency.

Flexibility and extensibility are an important part of good software. By giving extra thought to the way in which system data structures are to be organized, it should be possible to build a system that is flexible and open to many possible extensicns. The added work involved in developing these data structures is more than compensated for by the saving made if less adequate data structures were used and a new system had to be written at a later date due to of a lack of flexiblity.

Complete documentation consists cf three parts. The software specification is written before the programming begins and serves as a blueprint for the software. The effort put into a good specification is well worth-while

since it will probably prevent some over-looked item forcing major redesign after programming has begun. The Internal Logic Manual is a necessary document if other people are to understand, maintain and modify the software. Even the designer begins to forget how his software works. The User's Guide provides the users of the software with the necessary information to utilize the system.

The principles discussed in this thesis are not a complete list of software design principles. Much more research needs to be devoted to the formulation of general software design principles that are practical and usable. For example, the section on documentation in this thesis discussed several important aspects of documentation. However, it was beyond the scope of this thesis to develop a general-purpose format for the documentation of software packages of all types. This is one of the many things needed to advance the art of software design. Standards have been laid down and widely accepted for the definition of the FORTRAN language. The same should be done for a method of software documentation that could become as widely accepted as the definition of the FORTRAN language. If such a standard could be accepted and used, the problem of communication between software engineers could be reduced greatly. As a result, there would probably be much less duplication of software packages, which could release enormous resources, both in money and manpower, for the development of new software.

The area of documentation is just one example of how much more work is required so that a complete set of design software design principles could be formulated and eventually become the accepted standard for good software design.

The formulation and general acceptance of software design principles should greatly improve the quality and "life-span" of software packages.

# REFERENCES

Brooks F. P. and Iverson K. E., Automatic Data Processing, System /360 Edition, John Wiley and Sons, Inc., New York, 1969.

Control Data Corporation, 'GRID Graphic Terminal Subsystem Hardware Reference Manual', Data Display Division, Control Data Corporation, April 1969.

Gray M. and London K.R., 'Documentation Standards', Brandon/Systems Press Incorporated, 1969.

Hartman P.H. and Owens D.H., 'How to Write Software Specifications', AFIPS Conf. Proc., FJCC 31, pp 779-790, 1967.

Huen W.H., 'A Graphical Display Subroutine Package', Thesis, University of Alberta, 1969.

Jackson W.C., 'Computer Graphics for the Application Programmer', Department of Computing Science, University of Alberta, June 1972.

IBM, Form A22-6864-3, 'IBM 2701 Data Adaptor Unit Component Description', System Reference Library, July 1969.

IBM, Form C28-6514-7, 'IBM System/360 Operating System Assembler Language', System Reference Library, December 1970.

IBM, Form A22-6822-15, IBM System/360 and System/370 Bibliography, System Reference Library, July 1970.

IBM, Form C28-6535-7, 'IBM System/360 Operating System

Concepts and Facilities', System Reference Library, 1970

IBM, Form A22-6822-15, IBM System/360 and System/370 Biblicgraphy, System Reference Library, July 1970.

IBM, Form C28-6646-3, 'IBM System/360 Operating System Supervisor and Data Management Services, System Reference Library, 1970.

IBM, 'IEXT 360', Program Information Department, IBM Corporation, March 1969.

Lampson, B.W., 'On Reliable And Extendable Operating Systems', Software Engineering Techniques, NATC Science. Committee Conference Report, Edited by J.N. Buxton And B. Randell, April 1970.

Neumann P.G., 'The Role Cf Motherhood In The Pop Art Of System Programming', Second Symposium On Operating System Principles, Princeton University, Octcber 1969.

Sharp I.P., Software Engineering Techniques, Nato Science Committee Conference Report, Edited by J.N. Buxton And B. Randell, 1970.

Silver S.S., 'A Format Manipulation System For Autcmatic Production Cf Natural Language Documents', Institute Cf Library Research, University Of California, Los Angeles, Decemter 1969.

Tou, Julius T. (Editor), Software Engineering, Volumes I and II, Academic Press, New York, 1970.

UNIVAC, Division Of Sperry Rand Canada, 'Glcssary Of

Computer Terms', Sperry Rand Canada Ltd., 1959.

Walsh D., 'A Guide For Scftware Documentation',
Advanced Computer Techniques Corporation, 1969.

# APPENDIX I

## 3. FORTRAN Subroutines - The GRIDSUB Package

### 3.1 Introductory Description

A package of assembler language routines (GRIDSUB), callable from a FORTRAN program, is available for defining, displaying, and modifying graphic elements, and for controlling communication between the FORTRAN program and the display user. Concise definitions of these routines are given in Section 3.2. The routines are classified according to their functions:

(a) Block handling

(b) Definition of graphic elements

(c) Block display

(d) Display modification

(e) Transmission and message decoding

(f) Data conversion.

Examples of use of the routines are given in Section 5.

Any picture is defined as a set of discrete components called blocks, each with an identifying number in the range 1-511. A block definition is opened by a call to the subroutine BLOCK, and closed by a call to ENDBLK. The content of a block is defined by calls to the routines:

VECTOR, which defines a line,

DLINE, which defines a series of connected line segments,

MOVE, which defines a positicnal move without any line being drawn,

POINTX, which defines a pcint,

TEXT, which defines a string of alphanumeric characters.

The block definition is added to a file called the Block File. No display of the block can take place until a DISPLY call for that block is given, a DISPLY call must include the block's identification NUMBER. The block is the basic component for display and for light pen detection. For example, where a number of graphs are to be displayed, each with different axes, one graph with its axes could constitute a block. On the other hand, if several graphs are to be displayed all with the same axes, the axes could constitute a single separate block.

The blcck number can also be used to distinguish elements picked with the light pen. For example, where there are a number of command words to be displayed which must, after a light pen pick, be distinguishable frcm one another, each could be defined as a separate block. On the other hand, the words could all be defined within a single block, and the distinction between them made on the basis of their position or their ID.

Each blcck has an origin, the point at which definition begins. The programmer can define the coordinates of elements within the definition in either of two modes:

Absolute (with respect to the block origin)

Relative (with respect to the beam position after completing the previous element of the definition).

Selection from the block file of a defined block for actual display is effected with a call to the subroutine DISPLY. The block is referenced by its number, and a copy identification ID is added; the same block can be displayed, if required, with different ID's at up to 64 different positions on the screen with ID's in the range 0 to 63. Absolute screen coordinates are used to indicate the position on the screen of the block origin.

The DISPLY call causes the transfer of block definitions from the Block File to a buffer. The picture to be displayed can thus be assembled with a series of DISPLY calls. However the picture itself does not appear on the screen until the buffer(s) are transferred to the Display File in GRID with a TRANMT call.

The TRANMT subroutine sends the buffer(s) and any modifications which have been made to the Display File to GRID, and put the user's program into the "wait" state. No time is charged against the program until a message is received from the display operator. The operator can study the displayed image, and indicate the desired actions with the light pen and keyboard, normally one or more of the actions specified in Sections 2.3, 2.4 as being allowed under the MULTIBANK supervisor.

When the operator presses the SEND key to indicate "end of message", the application program resumes at the statement following the CALL TRANMT. At this point, the operator's message may be decoded by using special decoding routines:

| | |
|---|---|
| DLPEN | (to check for light pen picks) |
| DFKEY | (to check for status/function keys) |
| DALPHA | (to check for alpha characters) |
| DVECT | (to check for vectors) |
| DPOINT | (to check for points) |
| DEND | (to check for end-of-message) |

A call to any of these routines allows a specified message component to be checked to see if it is of the type expected. If the type is as expected, a branch is made to a statement number given as an argument, and, at the same time, parameters given in the CALL are set with the contents of the message component.

Processing of the message from the GRID operator will almost certainly show that a change to the picture is required. Even if, as will often happen, the operator has made a mistake, a diagnostic message will have to be displayed.

Blocks can be added to the picture with the DISPLY call. A position can be changed with a call to the MOVEBK subroutine. Blocks can be erased with the ERSBK subroutine.

Erased blocks can be made visible again with the reset routine, RSTBK.

ERSBK and RSTBK calls do not effect the Block File, nor do they cause removal of the definitions from the Display File (in the GRID core). They are therefore most suitable for temporary erasure. To conserve space in the GRID core, it may be advisable to use the FREEBK call, which deletes the definition from the Display File while retaining it in the Block File. DELID will destroy a copy of the block, again helping to conserve space in the Display File. With either FREEBK or DELID, the block or copy can be restored with a DISPLY call. However, where a block definition is no longer required-for example, initial messages to the operator are redundant after his first response - the subroutine DELBLK may be used to destroy the block definition in the Block File, as well as in the Display File. Destruction of unwanted block definitions helps to conserve space in the Block File.

Displayed blocks are normally light-pen detectable. It may be necessary, for example if blocks are overlaid on the screen, to vary the detectability of one or more copies of a block. This can be done with a call to the subroutine LP.

Subroutine TEXT allows definition of blocks including alphanumeric characters. However, the programmer may wish to display a result of a computation, in which case conversion from INTEGER or REAL format to character format is necessary

before TEXT can be used. Conversely, data input from the terminal is in character form, and must be converted to INTEGER or REAL before arithmetic can be done. The routines listed below, to save space, make use of the data conversion subroutines which are part of the FORTRAN formatted I/O routines.

| | |
|---|---|
| CHRFLT | (character to REAL) |
| INTCVT | (character to INTEGER) |
| FLTCHR | (REAL to character) |
| INTCHR | (INTEGER to character) |
| DMPHEX | (internal form to hexadecimal character) |

Finally, a dump routine is available for printing, in hexadecimal, the contents of all files and tables used by GRIDSUB. This routine may be useful if debugging proves difficult. The routine is called DUMPIT.

## 3.2 GRIDSUB Specifications

### 3.2.1 Error Handling

Errors detected by GRIDSUB result in an error message on SPRINT.

### 3.2.2 Block Definition Subroutines

#### (1) BLOCK (NUMBER)

Parameter NUMBER: Mode INTEGER[1]; Range 1≤NUMBER≤511

Function

The definition of a block is opened, and henceforth can be referred to by the value of NUMBER.

Possible Errors

1. Block NUMBER is not in range 1 to 511.
2. Block NUMBER is defined.
3. Previous block unended.
4. Block file is full.

#### (2) ENDBLK

Function

The definition cf a block is closed. Each definition must be closed with a call to ENDBLK before another definitien is opened.

[1]Mode INTEGER means INTEGER*4 hereafter unless otherwise specified.

Possible Errors

1. A block is not open.
2. Empty block, block not defined.
3. Block File full

### 3.2.3 Graphic Element Generation

(1) MOVE (RELBM, X, Y)

Parameters RELBM: Mode LOGICAL*1

X,Y: Mode INTEGER; Range -1023≤X,Y≤1023

Function

Within a block definition, the beam is moved without a line being drawn to a position:

(X'+X, Y'+Y) if RELBM is .TRUE.,

or (X,Y) if RELBM IS .FALSE., if the previous beam position was (X', Y') and where all coordinates are taken relative to the block origin.

Possible Errors

1. Block not opened.
2. X or Y increment not in range -1023 to 1023.
3. Block File full.

(2) VECTOR (RELBM, X, Y, BLINK, DASH)

Parameters RELBM: Mode LOGICAL*1

X,Y: Mode INTEGER; Range -1023≤X,Y≤1023

BLINK: Mode LOGICAL*1

DASH : Mode LOGICAL*1

Function

Within a block definition, a line is defined from the previous beam position to

(X'+X, Y'+Y) if RELBM is TRUE.

(X,Y) if RELBM is FALSF

where the previcus beam position was (X',Y') and where all coordinates are taken relative to the block origin. The lines will be shown blinking if BLINK is .TRUE., and dashed if DASH is .TRUE.

Possible Errors

1. Block not opened.
2. X or Y not in range -1023 to 1023.
3. Block File full.

(3) DLINE (RELBM, RELBM1, AX, AY, N, BLINK, DASH)

Parameters RELBM: Mode IOGICAL*1

RELBM1: Mode LOGICAL*1

AX,AY: Mode INTEGER*2 vector,Dimension=N

N: Mode INTEGER

Function

AX and AY are arrays of at least N elements. A line is defined through each of the N points represented by (AX(I), AY(I)), I = 1,N. The line begins at the point (AX(1), AY(1)).

If REIBM IS .TRUE., (AX(1),AY(1)) is taken to be the coordinates relative to the beam position before DLINE was called.

If REIBM is .FAISE., (AX(1), AY(1)) is taken to be the cocrdinates relative to the block origin.

If REIBM1 is .TRUE., (AX(I), AY(I)), 2≤I≤N, are taken tc be relative to the last beam position, that is as increments X , Y .

If REIBM1 is .FALSE., (AX(I), AY(I)), 2≤I≤N, are taken to be relative to the block origin.

The line will be shown blinking if BLINK is .TRUE., and dashed if DASH is .TRUE.

Possible Errors

1. Block not opened.
2. X or Y not in range -1023 to 1023.
3. Block File full.

(4) POINTX (RELBM, X, Y, BLINK)

Parameters RELBM: Mode LOGICAL*1

X,Y : Mode INTEGER, Range -1023≤X,Y≤1023

Function

Within a block definition, a point is defined at (X'+X, Y'+Y) if RELBM is .TRUE. or (X,Y) if RELBM is .FALSE. where (X',Y') is the previous beam positicn, and all coordinates are taken relative to the blcck origin. The point will be shown blinking if BLINK is .TRUE.

Possible Errors

1. BLOCK not opened.
2. X or Y not in range -1023 to 1023.
3. Block File full.

(5) TEXT (RELBM, X, Y, N, CHAR, LARGE, BLINK)

Parameters

RELBM: Mode LOGICAL*1

X,Y: Mode INTEGER; Range $-1023 \leq X,Y \leq 1023$

N: Mode INTEGER; Range $1 \leq |N| \leq 86$

CHAR: Mode: Any scalar or array, or Hollerith or literal string.

LARGE: Mode LOGICAL*1

BLINK: Mode LOGICAL*1

Function

Within a block definition, a string of |N| characters beginning at CHAR (or as the Hollerith or literal string) is defined. The position of the center of the first character is specified by RELBM, X, Y. If N is positive, the characters are defined in the +X direction; if N is negative, the characters are defined in the +Y direction.

The GRID character set differs slightly from that of the 029 punch (see Appendix I).

If LARGE is .TRUE., characters are approximately 13.6 (wide) x 17.9 (high) raster units, with spacing of 16 raster units between centres of successive characters.

If LARGE is .FALSE., characters are

approximately 10.7 (wide) x 14.2 (high) raster units, with spacing of 12 raster units.

Notes

(1) If, on display, the number of characters is greater than can be accommodated on one line, the characters will "wrap-around" without advancing to a new line.

(2) After definition of a character string, the beam position is assumed to be at the centre of the (undisplayed) symbol which would follow the last symbol of the string.

NOTE: 85 rasters=1 inch, 86 small characters/line, 64 large characters/line.

Possible Errors

1. Block not opened.
2. X or Y not in range -1023 to 1023
3. Number of characters not in range |1| to |86|.
4. Block File full.

(6) IDY (ID)

Parameter ID: Mode INTEGER; Range 0≤ID≤63

Function

The value of the copy identifier is set to the value given for ID (see the DISPLY routine). The value of ID returned after a light pen hit on the section of the block defined after the CALL IDY will have the value defined in the CALL IDY. (see the decoding routines).

Possible Errors:

1. Block not opened.
2. ID not in range 0 to 63.
3. Block File full.

Note

When issuing calls such as DELID, LP, MCVFBK etc., the ID GIVEN SHOULD BE THAT used in the DISPLY call.

3.2.4 Block Display

DISPLY (NUMBER, ID, ABSX, ABSY)

Parameters NUMBER: Mode INTEGER; Range

1≤NUMBER≤511

ID: Mode INTEGER; Range 0≤ID≤63

ABSX, ABXY: Mode INTEGER; Range 0≤ABSX, ABSY≤1023

Function

A previously defined block designated by NUMBER is transferred to the Display File for display. It will appear with its origin at (ABSX,ABSY). The copy of the block will be given the identification ID.

Note The values of ABSX and ABSY must be chosen so that no part of the block will be outside the screen limits.

Possible Errors

1. Block NUMBER not in range 1 to 511.
2. ID not in range 0 tc 63.
3. ABSX or ABSY not in range 0 to 1023.
4. Block to be displayed is not defined.
5. Block previously displayed with this ID - block not displayed.
6. Buffers full - block not displayed.
7. Bank Tables full - block not displayed.
8. Display file full - block not displayed.

### 3.2.5 Display Modification

(1) MOVEBK (NUMBER, ID, ABSX, ABSY)

Parameters NUMBER: Mode INTEGER; Range 1≤NUMBER≤511

ID: Mode INTEGER; Range 0≤ID≤63

ABSX: Mode INTEGER; Range 0≤ABSX≤1023

ABSY: Mode INTEGER; Range 0≤ABSY≤1023

Function

A displayed copy of a block designated by NUMBER and ID is moved to a new position (ABSX, ABSY), where ABSX and ABSY are in absolute screen coordinates.

Possible Errors

1. Block number not in range 1 to 511.
2. ID not in range 0 to 63.
3. ABSX or ABSY not in range 0 to 1023.
4. Block with specified ID is not currently being displayed.
5. Queue full - move not done.[1]

(2) ERSBK (NUMBER)

Parameter NUMBER: Mode INTEGER; Range 1≤NUMBER≤511

[1]Appendix III.

Function

All copies of the block designated by NUMBER are erased from the screen. The block definition remains in GRID core, and the block can be made visible again with the RSTBK subroutine.

Possible errors

1. Block number not in range 1 to 511.
2. Block to be erased is not displayed.
3. Queue full - erases not complete.[1]

(3) RSTBK (NUMBER)

Parameter NUMBER: Mode INTEGER; Range 1≤NUMBER≤511

Function

Copies of block NUMBER, previously erased by an ERSBK call, are made visible again.

Possible Errors

1. Block number not in range 1 to 511.
2. Block to be restored is not erased.
3. Queue full - restores not complete.[1]

(4) FREEBK (NUMBER)

Parameter NUMBER: Mode INTEGER; Range 1≤NUMBER≤511

Function

Block NUMBER is removed from the Display File. All copies will disappear from the screen. The block can be restored only by another DISPLY call.

Possible Errors

1. Block number not in range 1 to 511.
2. Block to be freed is not displayed.
3. Queue full - frees not complete.[1]

Note ERSBK and FREEBK:

ERSBK is faster and more convenient than FREEBK. FREEBK releases space in the Display File.

(5) DELID (NUMBER, ID)

Parameters NUMBER: Mode INTEGER; Range 1≤NUMBER≤511
ID: Mode INTEGER; Range 0≤ID≤63

Function

The copy ID of block NUMBER is deleted from the Display File. The copy will disappear from the screen. It can be restored only by

another DISPLY call.

Possible Errors

1. Block number not in range 1 to 511.
2. ID not in range 0 to 63.
3. Copy of block to be deleted is not displayed.
4. Queue full - block not deleted.[1]

(6) DELBLK (NUMBER)

Parameter NUMBER: Mode INTEGER; Range 1≤NUMBER≤511

Function

All information concerning block NUMBER is deleted from the Display File, and the Block File. NUMBER can then be used in the definition of another block.

Possible Errors

1. Block number not in range 1 to 511.
2. Block to be deleted is not displayed.
3. Queue full - deletes not complete.
4. Block to be deleted is not defined.

(7) LP (NUMBER, ID, DETECT)

[1]Appendix III.

Parameters NUMBER: Mode INTEGER; Range 1≤NUMBER≤511

ID: Mode INTEGER; Range 0≤ID≤63

DETECT: Mode LOGICAL*1

Function

A displayed copy of a block is detectable with the light pen unless otherwise specified. If DETECT=.FALSE., copy ID of block NUMBER is made not detectable. If DETECT=.TRUE., the copy's detectability will be restored.

Possible Errors

1. Block number not in range 1 to 511.
2. Block ID not in range 0 to 63.
3. Block to be altered is not displayed.
4. Queue full - detectability not altered.[1]

Note Selective detectability may be necessary if items are overlaid on the screen.

### 3.2.6 Transmission and Decoding Routines

(1) TRANMT

[1]Appendix III.

Parameters None.

Function

The buffer(s) are transmitted to GRID and the picture is displayed on the screen. The FORTRAN program then goes into the wait state until a message is received from the operator. The program resumes at the statement following the TRANMT call.

On the resumption, the FORTRAN program can decode the message using the decoding routines whose descriptions follow.

(2)

```
DLPEN     (INTNO, IX, IY, TYPE, ID, BLK, &N)
DFKEY     (INTNO, STATUS, KEY, &N)
DALPHA    (INTNO, IX, IY, NUM, MAX, STRING, &N)
DVECT     (INTNO, NUM, MAX, X, Y, &N)
DPOINT    (INTNO, NUM, MAX, X, Y, &N)
DEND      (INTO, &N)
```

Parameters

INTNO: Mode INTEGER; Range 1≤INTNO≤20

IX, IY: Mode INTEGER; Range 0≤IX,IY≤1023

TYPE: Mode INTEGER; Range 0≤TYPE≤2

ID: Mode INTEGER; Range 0≤ID≤63

BIK: Mode INTEGER vector; Dimension 8

STATUS: Mode INTEGER; Range 1≤STATUS≤15

KEY: Mode INTEGER; Range -1≤KEY≤9

NUM: Mode INTEGER; Range 1≤NUMBER≤MAX

MAX[1] Mode INTEGER; Range 1≤MAX≤43

STRING: Mode INTEGER*2 vector, Dimension MAX[1]

X: Mode INTEGER*2 vector, Dimension MAX[1]

Y: Mode INTEGER*2 vector, Dimension MAX[1]

&N: N is a statement number.

Function

A check is made to see if the i component (i specified by INTNC) is of the type:

light pen pick (DLPEN)
function key (DFKEY)
string of alpha characters (DALPHA)
string of vectors (DVECT)
string of points (DPOINT)
end of message (DEND)

If the type is as requested, control is transferred to the statement specified by n, with values supplied for the appropriate parameters. If the type is not as requested, control is transferred to the following statement (see example in Section 5.1.2).

Assuming that the action is correctly queried, then

[1]MAX chosen at the discretion of the programmer.

(i) With DIPEN,

IX, IY record positional information for the entity detected.

TYPE=0 if a point was picked,

TYPE=1 if a symbol was picked,

TYPE=2 if a vector was picked.

ID is set with the value given to the picked block with a DISPLY call.

BLK, which must be dimensioned as BLK(8), is set with the block number and nested block numbers of the entity picked with the pen. (Currently nested blocks are not supported so all elements except BLK(1) are zero).

BLK(1) is set with the number of the outermost block.

BLK(2) is set with the number of the first level nested block, and so on. Unused elements in BLK will be set to zero.

If, as would be most usual, what has been picked was not nested (by an ADDBLK), then only BLK(1) is set.

(ii) With DFKEY

STATUS is set with the value of the status key setting.

KEY is set with the value of the function key pressed, (or -1 if the INT key has

been pressed).

(iii) With DALPHA

IX, IY are set with the screen coordinates of the centre of the first character of the string.

STRING is set with the characters input, two to each half-word, left-to-right.

NUM is set with the number of half-words of STRING used. (The actual number of characters is 2*NUM or 2*NUM-1; all unused elements of STRING will be blanked).

MAX, the dimension of STRING, is supplied by the programmer to guard against a situation in which the operator types a string of characters longer than the space allowed in STRING.

(iv) With DVECT and DPOINT

Arrays X, Y are set with coordinates of the vector end points (DVECT), or points (DPOINT).

NUM is set with the number of points in each case, i.e. if NUM=n, then there are n-1 vectors or n points.

MAX, the dimension of X and Y, is supplied by the programmer. X(NUM+1) to X(MAX) and Y(NUM+1) to Y(MAX) will be set to zeros.

### 3.2.7 Data Conversion

(1) and (2) are function subroutines.

(3), (4) and (5) are subroutines to be called.

(1) X = CHRFLT (STRING, LENGTH, ERROR)

Parameters X: Mode: REAL or REAL*8

CHRFLT: Mode: Must be the same as for X

STRING: Mode: Any scalar, array, or literal string.

LENGTH: Mode INTEGER; Range 1≤LENGTH≤16

ERROR: Mode: LOGICAL*1

Function

X is set to the value of the floating-point number represented by the string of characters beginning at the left-most byte of STRING. The number of characters is specified by LENGTH. STRING may include a sign, or a decimal point. Absence of sign implies a positive number. If there is no decimal point, it is assumed to follow the string. The string may include leading blanks (only). ERROR is set to .TRUE.(and X to zero) if the string contains an illegal character, or if the LENGTH is not within the allowed range.

Otherwise, ERROR is set to .FALSE.

Examples:

(i) Z(I) = CHRFLT (P(J),8,L) sets Z(I) with the value of the number expressed in 8 characters beginning with the left-most byte of P(J).

(ii) Y= CHRFLT('-124.08',7,A)

(2) I = INTCVT (STRING, IENGTH, ERROR)

Parameters I: Mode: INTEGER or INTEGER*2

INTCVT: Mode: Must be the same as for I

STRING: Mode: Any scalar, array, or literal string.

LENGTH: Mode: INTEGER; Range 1≤LENGTH≤16

ERROR: Mode: LOGICAL*1

Function

I is set to the value of the integer number represented by the string of characters beginning at STRING. STRING may include leading blanks (only), a sign, but no decimal point. Absence of sign implies a positive number.

LENGTH and ERROR are as for CHRFLT.

Example M= INTCVT (A(I),10,E) is equivalent to

reading with I10 format.

(3) CALL FLTCHR (X,STRING,LENGTH,DECS)

Parameters X: Mode: REAL or REAI*8

STRING: Mode: Any scalar cr array

LENGTH: INTEGER variable or constant.

DECS: INTEGER variable or constant;

Range: 0≤DECS≤7

Function:

Floating-point number X is converted to a character string of length IENGTH, with DECS characters after the decimal pcint. If LENGTH is not sufficient, the string is filled with *'s. If LENGTH is greater than needed, leading blanks will be inserted.

Example:

CAII FLTCHR (X,Z,8,2) will convert the value of X to a character string beginning in the left-most byte of Z, with format conversion as F8.2.

(4) CALL INTCHR (J, STRING, I1, I2)

Parameters J: Mode: INTEGER or INTEGER*2

STRING: Mode: Any scalar cr array.

I1: Mode: INTEGER constant

I2: Mode: INTEGER constant

Function

Integer J is converted to a character string of length I2, beginning at the left-most byte of STRING. If there are more than I2 characters, STRING is filled with *'s. I1 must show the mcde cf J, 2 if J is INTEGER*2, 4 if J is INTEGER*4.

Example:

```
CALL INTCHR (M(I),P(J),2,8)
```

specifies that M(I), of mode INTFGER*2, is to be converted to an 8-character string, beginning at the first byte of F(J).

(5) CALL LMPHEX (STRING1, STRING2, LENGTH)

Parameters STRNG1: Mode: Any scalar cr array

STRNG2: Mode: Any scalar or array

LENGTH: Mode: INTEGER

Function

The contents of core storage beginning at STRNG1 is converted to a character string beginning at STRNG2, where STRNG2 becomes the character equivalent of the hexadecimal

representation. STRNG2 will be twice the length of STRNG1. LENGTH specifies the number of bytes of STRNG1.

Example:

CALL DMPHEX (A,B,128)

specifies that 128 bytes, beginning at the leftmost byte of A, must be converted to a 256 character string, beginning at the leftmost byte of B.

### 3.2.8 LOGICAL*1 Arguments

A useful convention is to give a declaration such as:

```
LOGICAL*1       A/.FALSE./,R/.TRUE./,BL/.TRUE./,NB/.FALSE./,
LG/.TRUE./,SM/.FALSE./,D/.TRUE./,ND/.FALSE./,LPCN/.TRUE./,
LPCFF/.FALSE./
```

to provide convenient mnemonics for those options to be specified by logical variables.

i.e. A for "absolute" (relative to block origin).

R for "retative" (to last beam position).

BL for "blinking".

NB for "not blinking".

LG for "large" characters.

SM for "small" characters.

D for "dashed" vector.

ND for "not dashed" vector.

LPON for "light pen detectable".

IPCFF for "not light pen detectable".

One can then, for example, give calls such as

CAIL VECTOR (A or R, X, Y, BL or NB, D or ND).

# APPENDIX II

## A Sample Program Using GRIDSUB

```
C THIS PROGRAM ALLOWS THE GRAPHICS OPERATOR TO DISPLAY UP
C TO SEVEN TRIANGLES AND/CR UP TC SEVEN SQUARES. THERE ARE
C THREE COMMANDS ON THE SCREEN: 'DELETE', 'TRIANGLE' AND
C 'SQUARE', TO CREATE A SQUARE CR A TRIANGLE, HE FIRST
C POINTS TO THE RELEVANT COMMAND WORD, THEN CREATES A POINT
C CN THE CRT TO DENOTE THE DESIRED POSITION, AND FINALLY
C PRESSES THE SEND KEY. HE MAY DELETE ALL THE TRIANGLES
C (OR SQUARES) BY FIRST POINTING TO 'DELETE' AND THEN TO
C 'TRIANGLE' (CR 'SQUARE'). HE ENDS THE PROGRAM BY PRESSING
C ANY STATUS KEY, FOLLOWED BY ANY FUNCTION KEY,AND, AS
C ALWAYS, THE SEND KEY. IF THE OPERATOR ERRS, A MESSAGE IS
C RETURNED: 'ERRCR TRY AGAIN'.
      SUBROUTINE GRAFIC
      IMPLICIT INTEGER(A-Z)
      INTEGER*2 XX,YY
      LOGICAI*1 T/.TRUE./,F/.FALSE./,FLAG/.FAISE./
      INTEGER BLK(8)
C  SET UP THE THREE COMMAND WOFDS
      CALL BLOCK(1)
      CALL TEXT(F,0,0,6,'DEIETE',F,F)
      CALL ENDBLK
      CALL BIOCK(2)
      CALL TEXT(F,0,0,8,'TRIANGIE',F,F)
      CALL ENDBLK
```

```
      CALL BLOCK(3)
      CALL TEXT(F,0,0,6,'SQUARE',F,F)
      CALL ENDBLK
C  SET UP THE ERROR MESSAGE
      CALL BLOCK(4)
      CALL TEXT(F,0,0,16,'ERROR--TRY AGAIN',T,I)
      CALL ENDBLK
C  CREATE A SQUARE
      CALL BLOCK(6)
      CALL VECTOR(F,200,0,F,F)
      CALL VECTOR(F,200,200,F,F)
      CALL VECTOR(F,0,200,F,F)
      CALL VECTOR(F,0,0,F,F)
      CALL ENDBLK
C  CREATE A TRIANGLE
      CALL BLOCK(5)
      CALL VECTOR(F,200,0,F,F)
      CALL VECTOR(F,100,200,F,F)
      CALL VECTOR(F,0,0,F,F)
      CALL ENDBLK
C  DISPLAY THE THREE COMMAND WORDS
      CALL DISPLY(1,1,5,850)
      CALL DISPLY(2,1,5,900)
      CALL DISPLY(3,1,5,950)
C  INITIALIZE ID'S FOR SQUARES AND TRIANGLES TO ZERO.
      SID=0
      TID=0
C  SEND DISPLAY FILE TO THE GRID UNIT
```

```
    1  CALL TRANMT
C   WE NOW HAVE A MESSAGE FROM THE OPERATOR
C   TEST WHETHER THE ERROR MESSAGE IS TO BE DELETED
C   FROM THE SCREEN
       IF(.NOT.FLAG) GO TO 2
C   THE ERROR MESSAGE WAS SENT LAST TIME, WE NOW DELETE IT.
       CALL FREEBK(4)
       FLAG=F
C   SEE IF FIRST OPERATOR ACTION WAS A FUNCTION KEY
C   DENOTING END OF JOB
    2  CALL DFKEY(1,STATUS,KEY,&4)
C   SEE IF IT WAS A LIGHT PEN HIT
       CALL DLPEN(1,X,Y,TYPE,ID,BLK,&5)
C   IT WAS NOT A FUNCTION KEY OR LIGHT PEN--
C   SEND ERROR MESSAGE TO GRID
    3  CALL DISPLY(4,1,400,500)
       FLAG=T
       GO TO 1
C   CHECK WHETHER LIGHT PEN WAS POINTED TO A COMMAND
C   IF NOT SEND ERROR MESSAGE
    5  IF(BLK(1).GT.3) GO TO 3
C   CHECK WHETHER IT WAS THE DELETE COMMAND
       IF(BLK(1).GT.1) GO TO 6
C   SERVICE THE DELETE COMMAND
C   CHECK IF SECOND ACTION WAS A LIGHT PEN HIT
C   IF NOT SEND ERROR MESSAGE
       CALL DLPEN(2,X,Y,TYPE,ID,BLK,&7)
       GO TO 3
```

```
C  CHECK IF PEN POINTED TO 'TRIANGLE' OR 'SQUARE'
   7  IF(BIK(1).EQ.3) GC TO 8
      IF(BLK(1).EQ.2) GO TO 9
      GO TO 3
C  DELETE ALL THE SQUARES
   8  CALL FREEBK(6)
      SID=0
      GO TO 1
C  DELETE ALI THE TRIANGLES
   9  CALL FREEBK(5)
      TID=0
      GO TO 1
C  SERVICE A TRIANGLE OR SQUARE COMMAND
C CHECK IF SECOND ACTION WAS A POINT--IF NOT SEND ERROR
   6  CALL DPOINT(2,NUM,1,XX,YY,&10)
      GO TO 3
C  CHECK WHETHER SQUARE OR TRIANGLE IS TO BE AIDED
   10 X=XX
      Y=YY
      IF(BLK(1).EQ.3) GO TO 11
C  ARE THERE AIREADY SEVEN TRIANGLES?
      IF(TID.GT.6) GO TO 3
C  INCREMENT ID FCR TRIANGLES
      TID=TID+1
C INSERT ANOTHER TRIANGLE
      CALL DISPLY(5,TID,X,Y)
      GO TO 1
C  ARE THERE ALREADY SEVEN SQUARES?
```

```
   11 IF(SID.GT.6) GO TO 3
C  INCREMENT ID FOR SQUARES
      SID=SID+1
C  INSERT ANOTHER SQUARE
      CALL DISPLY(6,SID,X,Y)
      GO TO 1
C  ERASE THE CRT AND SEND A FAREWELL TO INDICATE END OF JOB
    4 CALL SNAP(.05)
      CALL DELBLK(1)
      CALL DELBLK(2)
      CALL DELBLK(3)
      CALL DELBLK(5)
      CALL DELBLK(6)
      CALL BLOCK(7)
      CALL TEXT(F,0,0,8,'FAREWELL',I,T)
      CALL ENDBLK
      CALL DISPLY(7,1,400,500)
      CALL TRANMT
C  GIVE CONTROL BACK TO MASTER CONTROL ROUTINE
      RETURN
      END
```

# APPENDIX III

## OS S Calling Sequence

Program Entry

```
PRCGNAME CSECT
         STM     R14,R12,12(R13)   SAVE REGISTERS FOR RETURN
         LR      R12,R15           ESTABLISH BASE REGISTER
         USING   PROGNAME,R12      ESTABLISH ADDRESSABILITY
         LR      R10,R13           SAVE OLD SAVE POINTER
         LA      R13,SAVEAREA      LOAD SAVE POINTER
         ST      R10,4(R13)        STORE BACK POINTER
         ST      R13,8(R10)        STORE FORWARD POINTER
         B       START             GO TO START OF ROUTINE
SAVEAREA DC      18F'0'            SAVE AREA ALLOCATION
```

Program Exit

```
EXIT     L       R13,4(R13)        LOAD BACK POINTER
         LM      R14,R12,12(R13)   RELOAD REGISTERS
         SR      R15,R15           SET ZERO RETURN CODE
         BR      R14               RETURN TO CALLER
```

Save Area Format

| WORD | CONTENTS |
|---|---|
| 1 | Used by PL/1 programs |
| 2 | Address of previous save area (stored by calling program) |
| 3 | Address of next save area (stored by current program) |
| 4 | Register 14 (return address) |
| 5 | Register 15 (entry point address) |
| 6 | Register 0 |
| 7 | Register 1 (parameter list address) |
| 8 | Register 2 |
| 9 | Register 3 |
| 10 | Register 4 |
| 11 | Register 5 |
| 12 | Register 6 |
| 13 | Register 7 |
| 14 | Register 8 |
| 15 | Register 9 |
| 16 | Register 10 |
| 17 | Register 11 |
| 18 | Register 12 |

Register Conventions

Register 1- contains the address of a list of addresses which point to the respective parameters. The last address in the list has the high order bit set to 1.

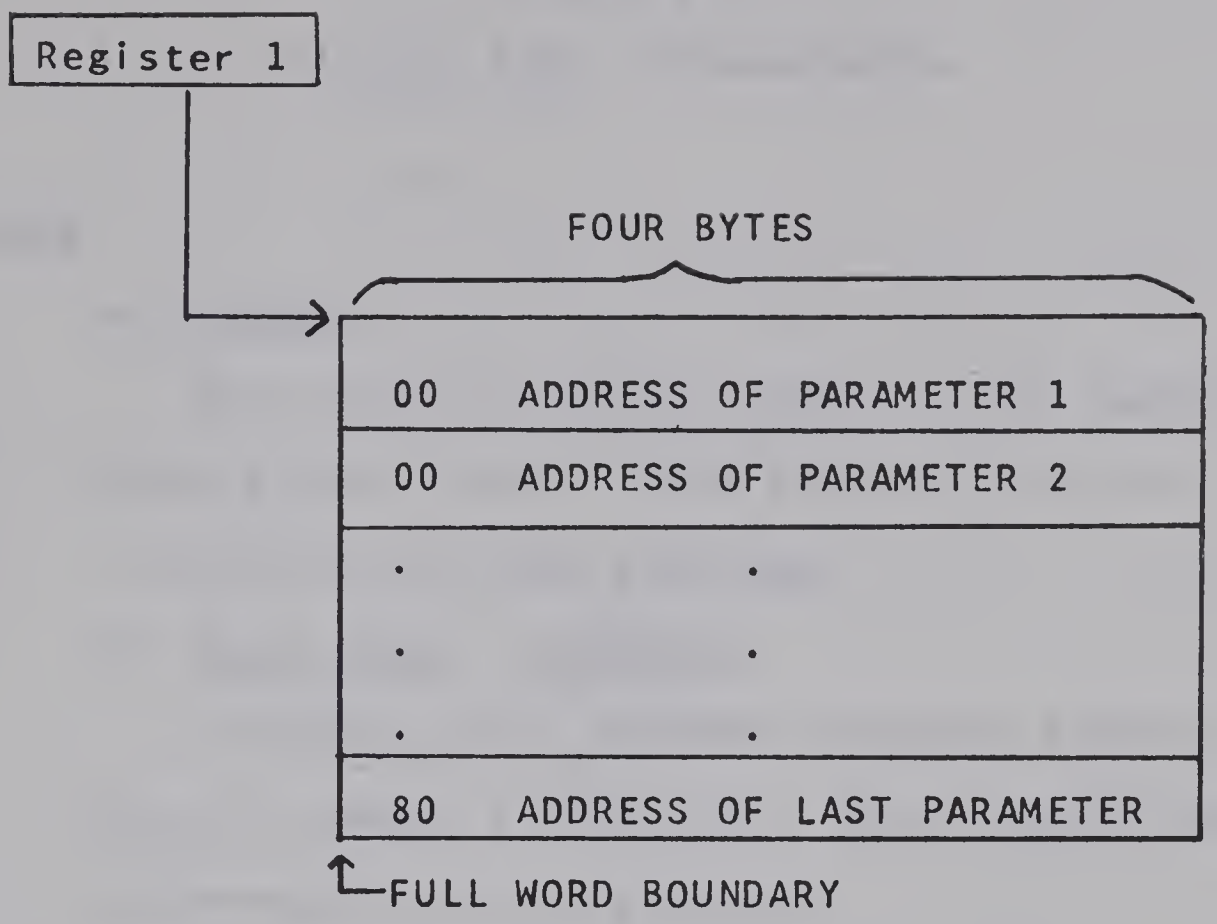



Figure III-1 Standard Parameter List Convention

Register 13 - contains the address of the save area used in processing requests made through system macros. This save area is also used to store the current programs registers if another program is called.

Register 14 - contains the address to which the current program is to return.

Register 15 - contains the entry point address when control is passed between routines. This register is used to pass a return code to the calling program when return is made.

# APPENDIX IV

## Internal Logic Documentation

### PACTOBIN

a) Purpose

This routine converts packed format data to binary format data. This routine does not reference any other routines.

B) Entry Point - BINTOPAC

The entry point BINTOPAC converts binary format data to packed format data. This routine does not reference any other routines.

C) Calling Routines

(I) PACTOBIN

TRANMT calls PACTOBIN to Unpack header and message received from GRID.

(II) BINTOPAC

TRANMT calls BINTOPAC to pack headers and messages before transmitting messages to GRID.

D) Calling Sequence

Both PACTOBIN and BINTOPAC use standard CS linkage conventions and require two parameters.

E) Parameters

Parameter 1 - full word containing the address of the text to be converted. address must be halfword aligned.

Parameter 2 - full word containing the

length in bytes of the text to be converted.

F) Exit Points

Both PACTOBIN and BINTOPAC return through the same exit point. Return code is always zero as no stock is made on the validity of the data.

G) Tables

No tables are referenced by either routine.

H) Processing Performed

In the packed format, two bytes in the 360 transferred through the interface correspond to one core memory location. For packed mode transmission the highest order bit in each byte must be 1 to avoid interface interpretation as a central character. The preparing of data for transmission and the conversion of received data is performed by BINTOPAC and PACTOBIN respectively.
See the flowchart and following diagrams.

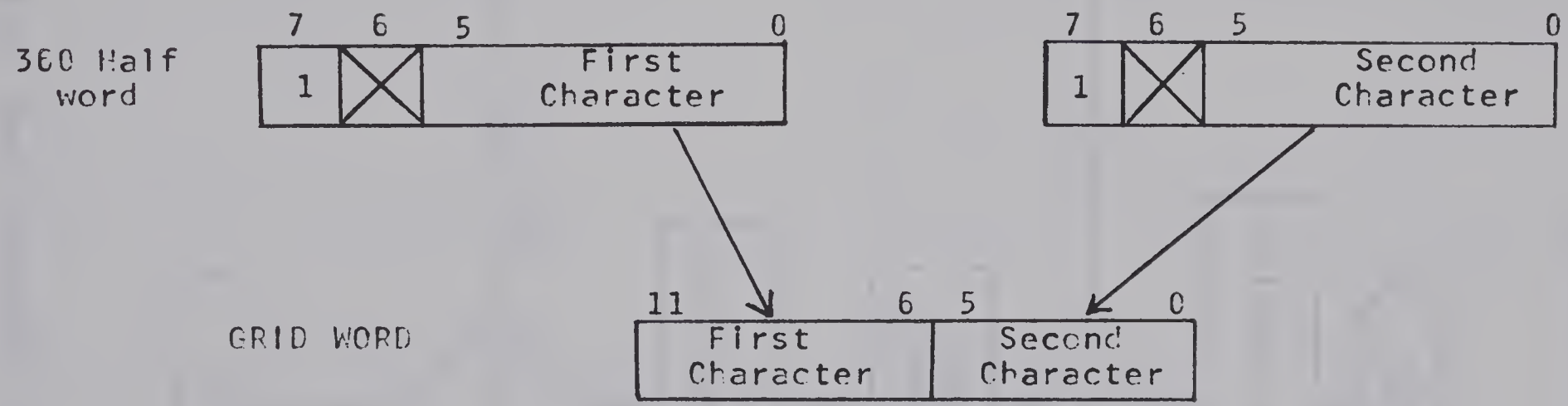



FIGURE IV-1 PACKED FORMAT

PACTOBIN/BINTOPAC

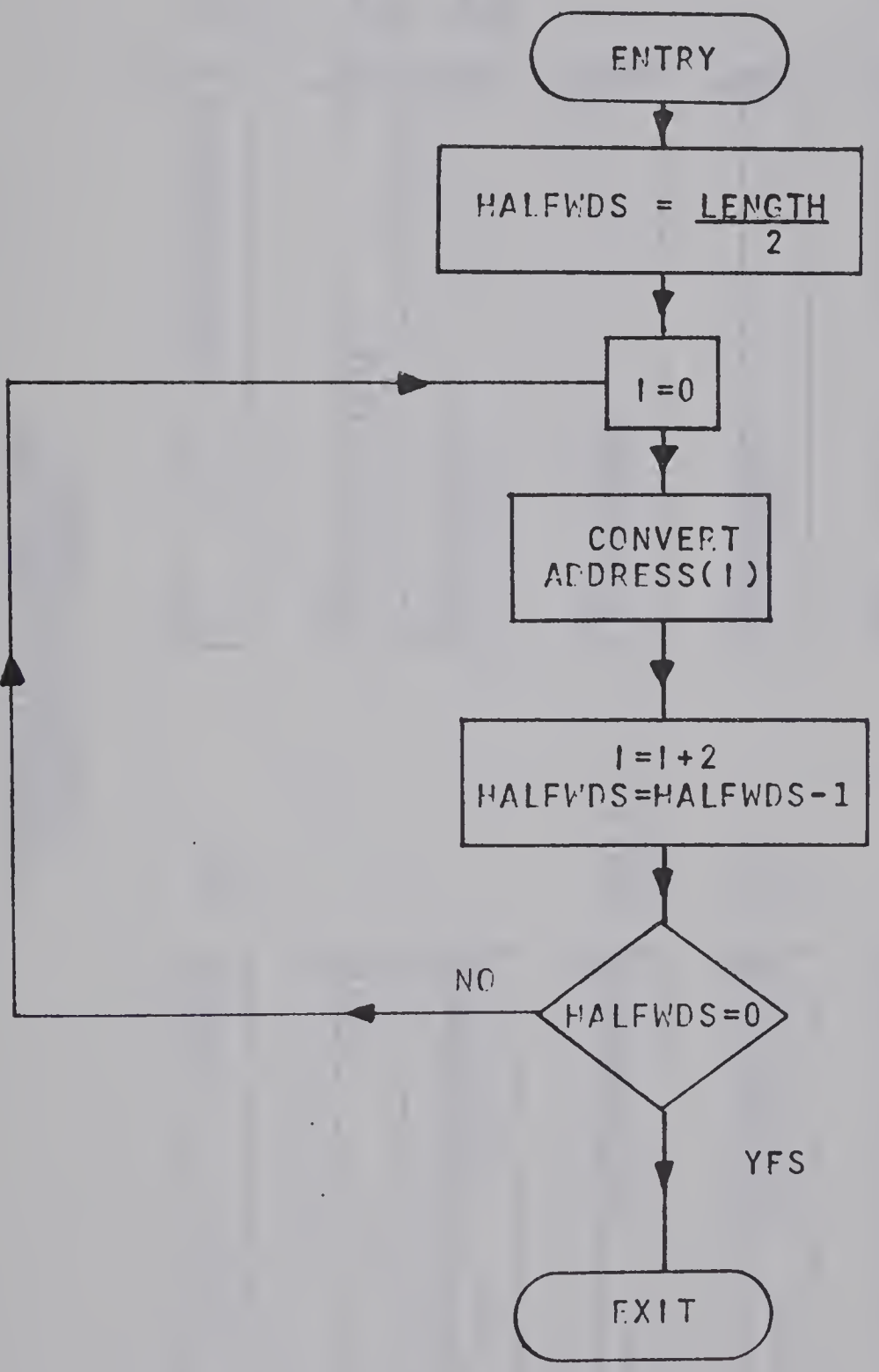



FIG. IV-2 FLOWCHART FOR PACTOBIN/BINTOPAC

## CONVERSION STEPS FOR BINARY TO PACKED

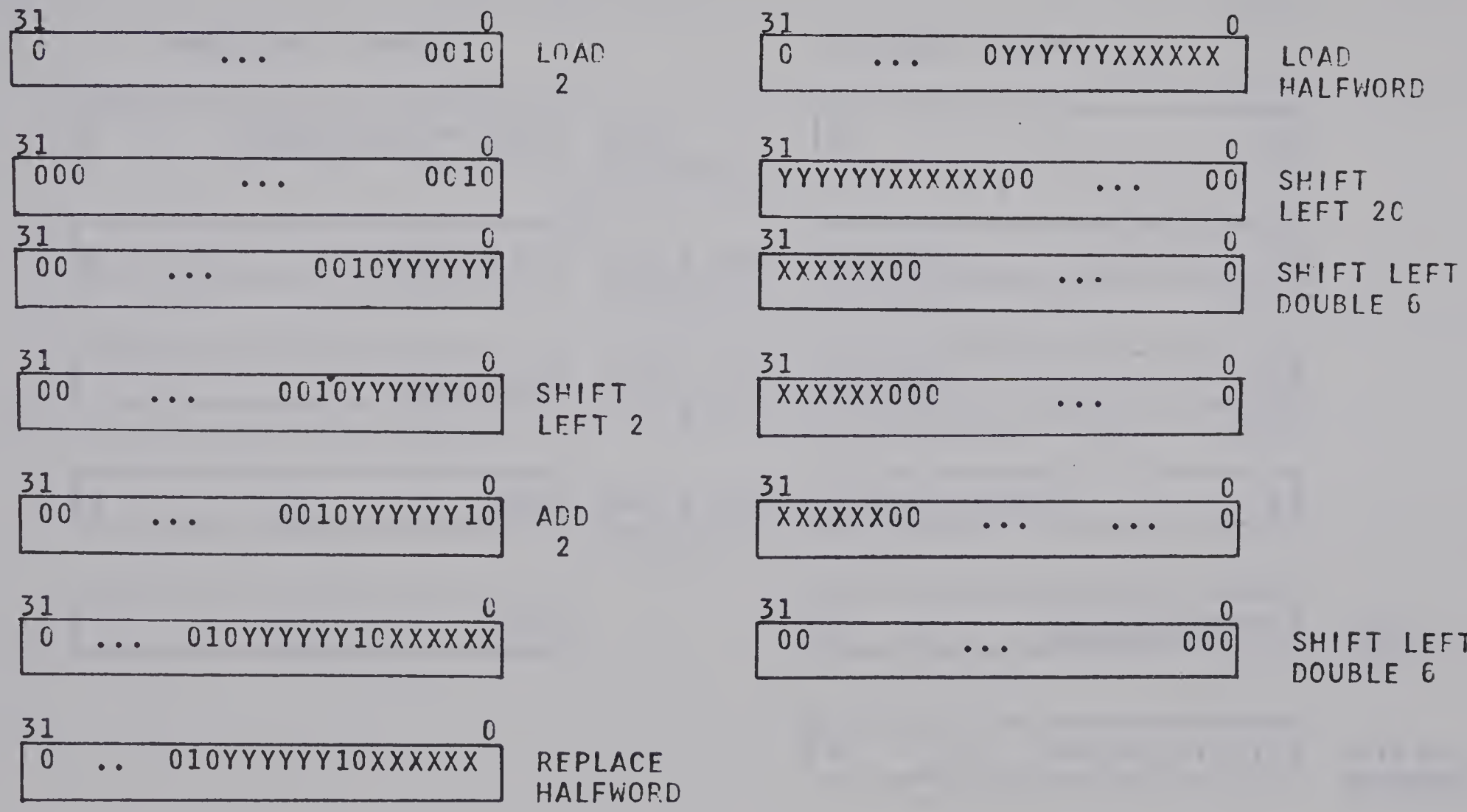



FIGURE IV-3 BINARY TO PACKED FORMAT

## CONVERSION STEPS FOR PACKED TO BINARY

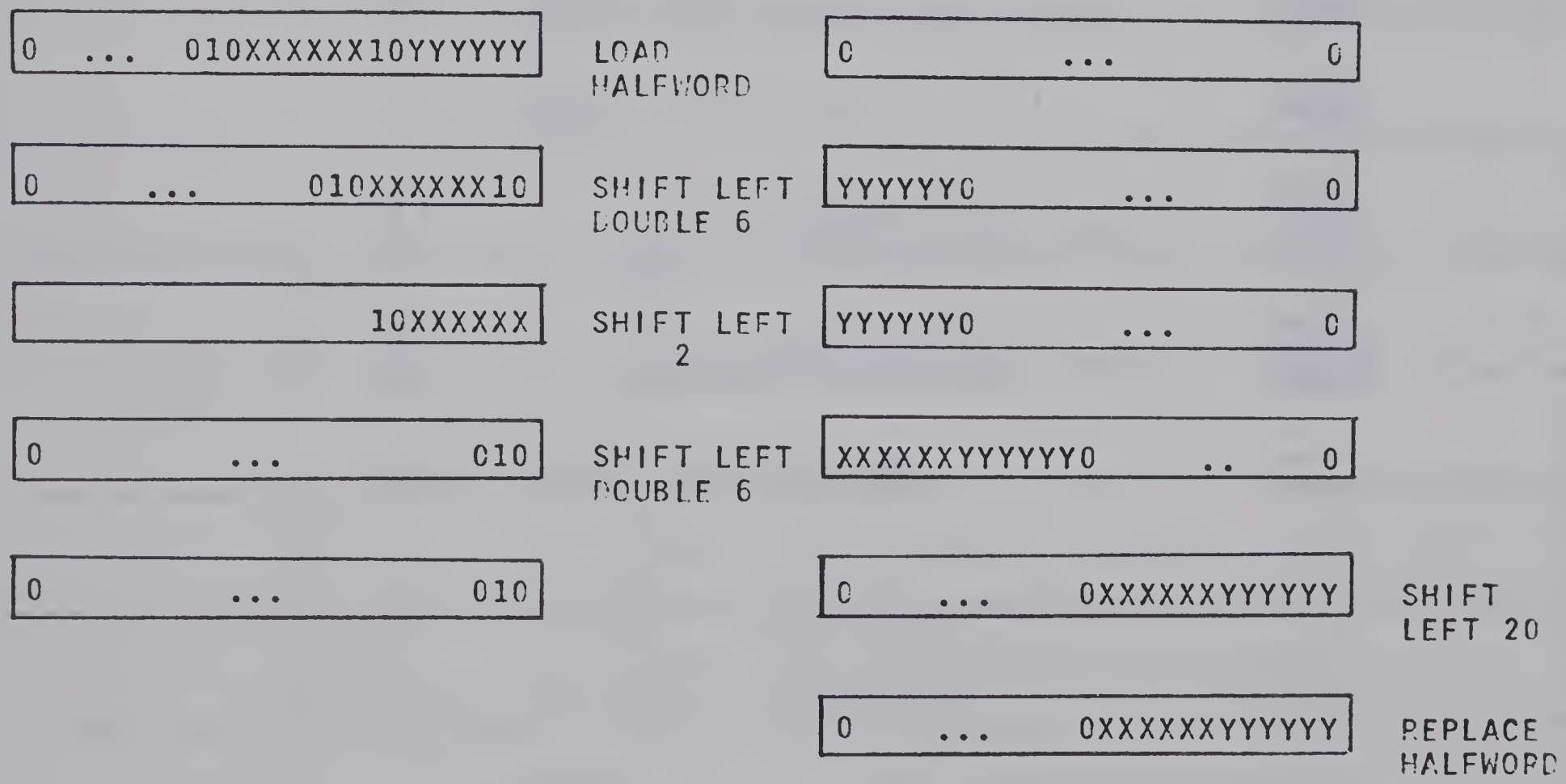



FIGURE IV-4 PACKED FORMAT TO BINARY

```
LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                  CONV0001
000000                             2 PACTOBIN CSECT                                       CONV0002
                                   3         ENTRY BINTOPAC
                                   4
                                   5 *       PARAMETER 1 - TEXT                           CONV0004
                                   6 *       ADDRESS OF TEXT TO BE CONVERT (HALFWORD ALIGNMENT) CONV0005
                                   7
                                   8 *       PARAMETER 2 - LENGTH                         CONV0007
                                   9 *       LENGTH OF TEXT TO BE CONVERTED               CONV0008
                                  10
                                  11 *       15              8|7             0            CONV0010
                                  12 *       ---------------------------------            CONV0011
                                  13 *       |0|0|0|0|X|X|X|X|X|Y|Y|Y|Y|Y|Y|  BINARY     CONV0012
                                  14 *       ---------------------------------            CONV0013
                                  15
                                  16 *       15              8|7             0            CONV0015
                                  17 *       ---------------------------------            CONV0016
                                  18 *       |1|0|X|X|X|X|X|X|1|0|Y|Y|Y|Y|Y|Y|  PACKED     CONV0017
                                  19 *       ---------------------------------            CONV0018
                                  20
                                  21 *       BINTOPAC - BINARY TO PACKED FORMAT           CONV0020
                                  22 *       PACTOBIN - PACKED FORMAT TO BINARY           CONV0021
```

```
LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                    22 JUL 71
000000                             24 R0       EQU   0                                                  CONV0023
000001                             25 R1       EQU   1                                                  CONV0024
000002                             26 R2       EQU   2                                                  CONV0025
000003                             27 R3       EQU   3                                                  CONV0026
000004                             28 R4       EQU   4                                                  CONV0027
000005                             29 R5       EQU   5                                                  CONV0028
000006                             30 R6       EQU   6                                                  CONV0029
000007                             31 R7       EQU   7                                                  CONV0030
000008                             32 R8       EQU   8                                                  CONV0031
000009                             33 R9       EQU   9                                                  CONV0032
00000A                             34 R10      EQU   10                                                 CONV0033
00000B                             35 R11      EQU   11                                                 CONV0034
00000C                             36 R12      EQU   12                                                 CONV0035
00000D                             37 R13      EQU   13                                                 CONV0036
00000E                             38 R14      EQU   14                                                 CONV0037
00000F                             39 R15      EQU   15                                                 CONV0038

000000 90EC D00C      0000C        41          STM   R14,R12,12(R13)    SAVE REGS FOR RETURN            CONV0040
000004 18CF                        42          LR    R12,R15            ESTABLISH BASE REG              CONV0041
000006                             43          USING PACTOBIN,R12                                         CONV0042
000006 18AD                        44          LR    R10,R13                                            CONV0043
000008 4100 C018      00018        45          LA    R13,SAVEAREA       LOAD SAVE POINTER               CONV0044
00000C 50DA C008      00008        46          ST    R13,8(R10)         STORE FORWARD POINTER           CONV0045
000010 50AD C004      00004        47          ST    R10,4(R13)         STORE BACK POINTER              CONV0046
000014 47F0 C06C      00060        48          B     START                                              CONV0047
000018 0000000000000000           49 SAVEAREA DC    18F'0'                                             CONV0048

00001C 5821 0000      00000        51 START    L     R2,0(R1)           LOAD ADDRESS OF TEXT            CONV0050
000020 5831 0004      00004        52          L     R3,4(R1)           LOAD ADDRESS OF LENGTH          CONV0051
000024 5833 0000      00000        53          L     R3,0(R3)           LOAD LENGTH                     CONV0052
000028 8A30 0001      00001        54          SRA   R3,1               CALCULATE NUMBER OF HALFWORDS   CONV0053
00002C 4842 0000      00000        55 LOOP     LH    R4,0(R2)           LOAD HALF WORD OF TEXT          CONV0054
000030 1855                        56          SR    R5,R5              ZERO REG 5                      CONV0055
000032 8C40 0006      00006        57          SRDL  R4,6               SHIFT TO REG 5 FIRST SIX BITS   CONV0056
000036 8840 0002      00002        58          SRL   R4,2               DROP BITS 6 AND 7               CONV0057
00003A 8C40 0006      00006        59          SRDL  R4,6               SHIFT TO R5 BITS 8 TO 13        CONV0058
00003E 8850 0014      00014        60          SRL   R5,20              PLACE THE 12 BITS IN LOWER ORDER OF REG 5  CONV0059
000042 4052 0000      00000        61          STH   R5,0(R2)           RESTORE HALFWORD IN BINARY FORM  CONV0060
000046 4122 0002      00002        62          LA    R2,2(R2)           INCREMENT POINTER               CONV0061
00004A 4630 C030      00030        63          BCT   R3,LOOP            BRANCH UNTIL COMPLETE           CONV0062

                                   65 *        RETURN TO CALLER                                           CONV0064

00004E 58D0 D004      00004        67 EXIT     L     R13,4(R13)         RESTORE BACK POINTER            CONV0066
000052 98EC D00C      0000C        68          LM    R14,R12,12(R13)    RELOAD REGS                     CONV0067
000056 18FF                        69          SR    R15,R15            SET RC=0                        CONV0068
000058 07FF                        70          BR    R14                RETURN                          CONV0069
00005A                             71          LTORG                                                    CONV0070
```

22 JUL 71

```
LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                    CONV0072
0000A0 90EC D00C             000C    73 BINTOPAC STM   R14,R12,12(R13)    SAVE REGS FOR RETURN            CONV0073
0000A4 18CF                          74          LR    R12,R15            ESTABLISH BASE REG              CONV0074
0000A6                               75          USING BINTOPAC,R12                                         CONV0075
000000                               76          USING PACTOBIN,R11                                         CONV0076
0000A6 58B0 C058             00F8    77          L     R11,=V(PACTOBIN)                                     CONV0077
0000AA 18AD                          78          LR    R10,R13                                              CONV0078
0000AC 41D0 B018             0018    79          LA    R13,SAVEAREA   LOAD SAVE POINTER                     CONV0079
0000B0 50DA 0008             0008    80          ST    R13,8(R10)     STORE FORWARD POINTER                 CONV0080
0000B4 50AD 0004             0004    81          ST    R10,4(R13)     STORE BACK POINTER                    CONV0081
0000B8 5821 0000             0000    82          L     R2,0(R1)       LOAD ADDRESS OF TEXT                  CONV0082
0000BC 5831 0004             0004    83          L     R3,4(R1)       LOAD ADDRESS OF LENGTH                CONV0083
0000C0 5833 0000             0000    84          L     R3,0(R3)       LOAD LENGTH                           CONV0084
0000C4 8A30 0001             0001    85          SRA   R3,1           BYTES/2 = HALFWORDS                   CONV0085
0000C8 4852 0000             0000    86 AGAIN    LH    R5,0(R2)       LOAD HALF WORD OF TEXT                CONV0086
0000CC 4140 0002             0002    87          LA    R4,2           PLACE  A 10 IN REG 4                  CONV0087
0000D0 8950 0014             0014    88          SLL   R5,20          SHIFT LOW 12 BITS TO HIGH ORDER POSITION CONV0088
0000D4 8D40 0006             0006    89          SLDL  R4,6           EXTRACT 6 BITS                        CONV0089
0000D8 8940 0002             0002    90          SLL   R4,2           SHIFT TWO PLACE FOR 10                CONV0090
0000DC 4144 0002             0002    91          LA    R4,2(R4)       SET 10 IN LOW BITS                    CONV0091
0000E0 8D40 0006             0006    92          SLDL  R4,6           EXTRACT LAST 6 BITS                   CONV0092
0000E4 4042 0000             0000    93          STH   R4,0(R2)       RESTORE HALFWORD IN PACKED FORMAT     CONV0093
0000E8 4122 0002             0002    94          LA    R2,2(R2)       INCREMENT POINTER                     CONV0094
0000EC 4630 C0C8             00C8    95          BCT   R3,AGAIN                                             CONV0095
0000F0 47F0 B092             0092    96          B     EXIT           RETURN TO CALLER                      CONV0096
0000F8                               97          LTORG                                                      CONV0097
0000F8 00000000                      98                =V(PACTOBIN)
                                     99          END
```

22 JUL 71

| SYMBOL | LEN | VALUE | DEFN | REFERENCES | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AGAIN | 4 | 0000C8 | 86 | 95 | | | | | | | | | | | | |
| BINTOPAC | 4 | 0000A0 | 73 | 3 | 75 | | | | | | | | | | | |
| EXIT | 4 | 000092 | 67 | 96 | | | | | | | | | | | | |
| LOOP | 4 | 000070 | 55 | 63 | | | | | | | | | | | | |
| PACTOBIN | 1 | 000000 | 2 | 43 | 76 | | | | | | | | | | | |
| R0 | 1 | 000000 | 24 | | | | | | | | | | | | | |
| R1 | 1 | 000001 | 25 | 51 | 52 | 82 | 83 | | | | | | | | | |
| R10 | 1 | 00000A | 34 | 44 | 46 | 47 | 78 | 80 | 81 | | | | | | | |
| R11 | 1 | 00000B | 35 | 76 | 77 | | | | | | | | | | | |
| R12 | 1 | 00000C | 36 | 41 | 42 | 43 | 68 | 73 | 74 | 75 | | | | | | |
| R13 | 1 | 00000D | 37 | 41 | 44 | 45 | 46 | 47 | 67 | 67 | 68 | 73 | 78 | 79 | 80 | 81 |
| R14 | 1 | 00000E | 38 | 41 | 68 | 70 | 73 | | | | | | | | | |
| R15 | 1 | 00000F | 39 | 42 | 69 | 69 | 74 | | | | | | | | | |
| R2 | 1 | 000002 | 26 | 51 | 55 | 61 | 62 | 62 | 82 | 86 | 93 | 94 | 94 | | | |
| R3 | 1 | 000003 | 27 | 52 | 53 | 53 | 54 | 63 | 83 | 84 | 84 | 85 | 95 | | | |
| R4 | 1 | 000004 | 28 | 55 | 57 | 58 | 59 | 87 | 89 | 90 | 91 | 91 | 92 | 93 | | |
| R5 | 1 | 000005 | 29 | 56 | 56 | 60 | 61 | 86 | 88 | | | | | | | |
| R6 | 1 | 000006 | 30 | | | | | | | | | | | | | |
| R7 | 1 | 000007 | 31 | | | | | | | | | | | | | |
| R8 | 1 | 000008 | 32 | | | | | | | | | | | | | |
| R9 | 1 | 000009 | 33 | | | | | | | | | | | | | |
| SAVEAREA | 4 | 000018 | 49 | 45 | 79 | | | | | | | | | | | |
| START | 4 | 000060 | 51 | 48 | | | | | | | | | | | | |

NO STATEMENTS FLAGGED IN THIS ASSEMBLY
13:01.13 4.13 RC=0

# APPENDIX V

## FILE AND TABLE USAGE UNDER GRIDSUB

A programmer using the GRIDSUB package may occasionally wish to know how close he is to exceeding the limits of the system. The capacities of files, and of tables which index these files, are given below:

(1) BLOCK FILE (40K bytes)

(2) BANK 0 TABLE - 1 entry
BANK 1 TABLE - 256 entries
BANK 2 TABLE - 256 entries

(3) DISPLAY FILE (SPACE AVAILABLE FOR CODE IN GRID CODE)
- approx. 8000 GRID words.

(4) MESAGE - 400 bytes (approx. 200 grid words)

(5) QUEUE - 100 entries (50 if entries cause by MOVEBK)

(6) BUFFERS (2) - each 2048 bytes

The sizes of these tables may be changed by re-assembling various portions of the GRIDSUB system. (See system description) The Display File space uses virtually all of Banks 1 and 2 and cannot be increased. Bank 0 is currently dedicated to the supervisor which may be readily expanded.

The function of each file and table is as follows:

BLOCK FILE - storage space for GRID code resulting from Block definitions.

BANK TABLES - describe the contents of the banks, one entry for every copy of a Block in the bank and one or more entries describing the supervisor.

DISPLAY FILE - space reserved in GRID core for the user's display code (virtually all of Banks 1 and 2).

MESAGE - a CSECT into which the message from GRID is placed. Both this CSECT and MESSAGE in the supervisor must be changed to alter message length.

QUEUE - used to keep track of changes that need to be made to the current display file. All entries are removed each time TRANMT is called.

If a call to a display modification routine results in the error message 'Queue Full ... ' unpredictable errors may result. If this error occurs, the user should either:

1. Make fewer display modification calls before calling TRANMT.
2. Reassemble the GRIDSUB module QUEUE giving it more space.

BUFFERS - to store code that is to be sent to GRID. The size of the buffers and their number should not be changed. If Buffer over-flow occurs, use more calls to TRANMT and make fewer additions to the Display File between calls to TRANMT. The buffers are emptied

each time TRANMT is called.

In the Block File entries are created by a combination of calls to the MOVE, VECTOR, POINT, DLINE, TEXT, BLOCK, ENDBLK, and IDY routines; entries are deleted by the DELBLK routine. In the Display File entries are created by the DISPLAY routine; and entries are deleted by the FREEBK, DELBLK, and DELID routine.

Comments and Suggestions for storage economy:

Most economic use of storage can be achieved if

1. Calls of the same type can be grouped together.
2. Beam movements can be kept small.
3. If the Display File is nearly full, and there are some Blocks in the Display File which have been 'erased' (not deleted), use the call FREEBK on these erased Blocks. This will release storage in DSFILE without destroying the Block definition in the BLFILE. The Block can be displayed again with a CALL DISPLY.
4. If a Block is finished with, make sure that DELBLK, rather than ERSBK, is called.

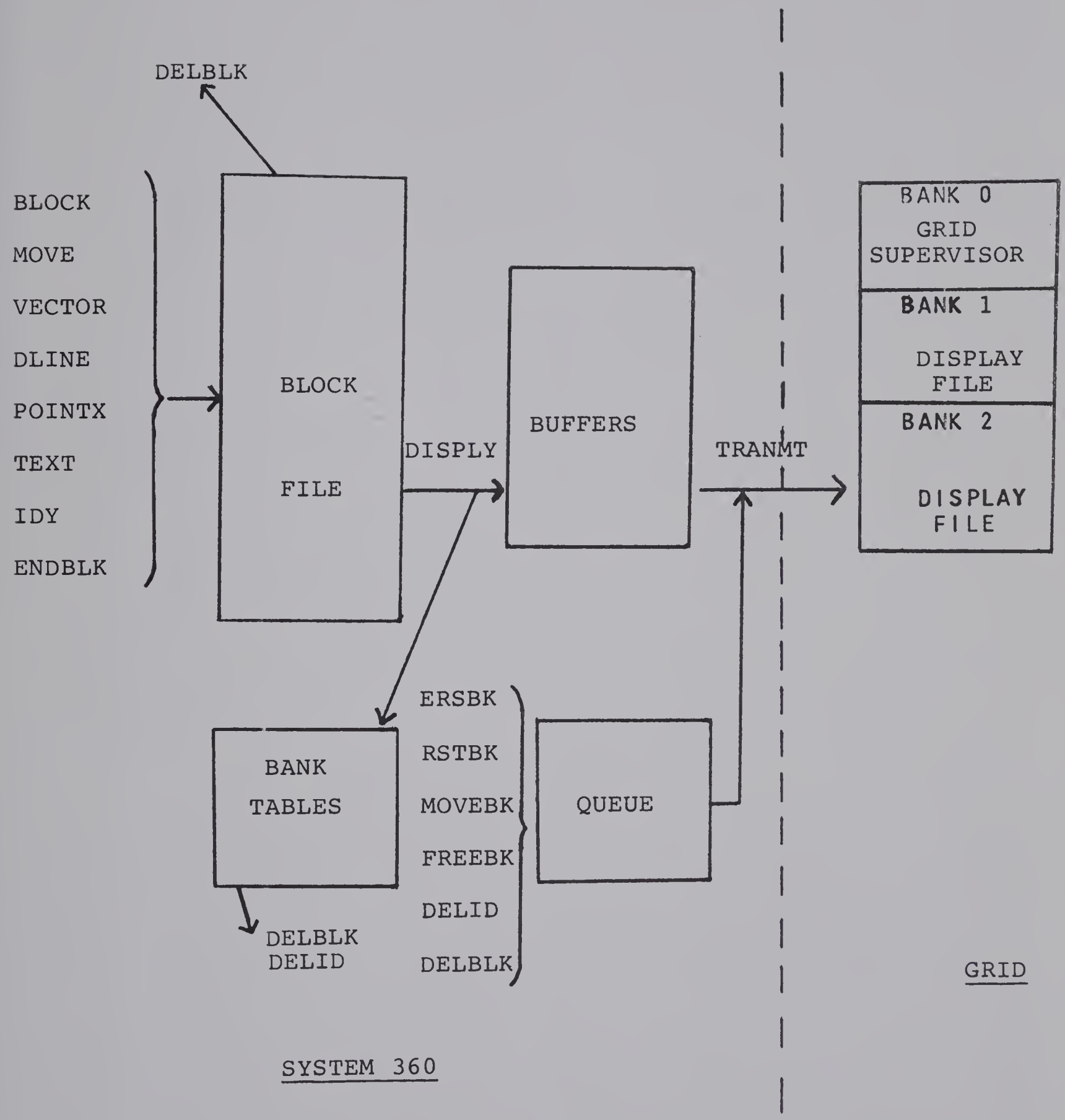



FIGURE V-1 GRIDSUB